Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1992-03

Implementing relational operations in an object-oriented database

Filippi, Stephen Charles

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/23699

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**IMPLEMENTING RELATIONAL OPERATIONS
IN AN
OBJECT-ORIENTED DATABASE**

by

Stephen C. Filippi

March 1992

Thesis Advisor:	Michael L. Nelson
Co-Advisor:	C. Thomas Wu

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School

6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
IMPLEMENTING RELATIONAL OPERATIONS IN AN OBJECT-ORIENTED DATABASE (U)

12. PERSONAL AUTHOR(S)
Filippi, Stephen Charles

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM _____ TO:_____	14. DATE OF REPORT (Year, Month, Day) 1992, March	15. PAGE COUNT 176

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	OBJECT-ORIENTED, DATABASES, RELATIONAL DATABASES

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis expands the concepts of relational/object-oriented database systems. There are two different approaches to combining relational and object-oriented databases. This thesis takes the approach of adding relational operations to an object-oriented database rather than building an object-oriented layer on top of an existing relational database.

The system proposed in this thesis was developed in the object-oriented programming language Prograph. It was chosen because it contains primitive operations for reading and writing database files to secondary storage and for manipulating complex data types (e.g., sounds, and pictures).

This thesis demonstrates that the limitations of current systems can be remedied and that the relational/object-oriented database management system is indeed a feasible solution.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Maj. Michael L.Nelson, USAF	22b. TELEPHONE (Include Area Code) (408) 646-2026	22c. OFFICE SYMBOL CS/Ne

D FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete   UNCLASSIFIED

i

Implementing Relational Operations
in an
Object-Oriented Database

by
Stephen Charles Filippi
Lieutenant, United States Navy
B.S., Jacksonville University, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

ABSTRACT

This thesis expands the concepts of relational/object-oriented database systems. There are two different approaches to combining relational and object-oriented databases. This thesis takes the approach of adding relational operations to an object-oriented database rather than building an object-oriented layer on top of an existing relational database.

The system proposed in this thesis was developed in the object-oriented programming language Prograph. It was chosen because it contains primitive operations for reading and writing database files to secondary storage and for manipulating complex data types (e.g., sounds, and pictures).

This thesis demonstrates that the limitations of current systems can be remedied and that the relational/object-oriented database management system is indeed a feasible solution.

Table of Contents

LIST OF FIGURES

ACKNOWLEDGEMENTS

This thesis was made possible through the efforts of many people. Only a few will be specifically mentioned here, but if you were involved in my research and your name isn't here, thank you.

First and foremost, thanks to my advisors Dr. Nelson and Dr. Wu. Had it not been for their challanging questions and patient guidance this research would have never been finished.

Special thanks goes to Lynn McKaig of TGSSystems. Without her prompt response to my incredibly obvious questions, I'd still be looking for solutions. Also her willingness to send me sample classes to clarify difficult points was most invaluable.

A warm thank you goes to my many friends at Apple Computer, Inc. Especially Carmela Zamora and Craig Elliot. Thanks for getting me E.T.O. I regret that it didn't get used in this research.

And for the most important people in my life, my family, without whom I certainly would not be where I am today. Thanks for all the encouragement, love, and support, especially from my wife Delilah. I only wish that I had been able to spend more time with you. And finally a special thank you to my daughters, Courtney and Gabrielle for making me smile even when my mind was pre-occupied.

I. INTRODUCTION

The purpose of this thesis is to expand upon the concepts of relational/object-oriented database systems. Much research is being conducted in the area of database technology. Most research, however, centers around query optimization, distributed databases, multimedia databases, and expert systems [EN89]. Some research is being conducted in the area of object-oriented (OO) databases; however, most of this research appears to be directed towards designing OO databases for specific types of applications such as computer aided software engineering (CASE) and computer aided design (CAD) tools.

There has been some research done in the area of combining relational database technology with object-oriented databases, and, two separate models have been proposed. The first approach states that to achieve a relational/object-oriented database management system (R/OODBMS), the object-orientation should be built on top of an existing relational database [PBRV90]. Specifically, an object-oriented interface is created to mask the underlying relational database. This interface appears to perform queries, updates, etc. on objects, but it actually performs the operations on the data that is stored in the relational database management system (R/DBMS) to manipulate the data. The other school of thought is to add relational operations to an object-oriented database [Nels88, NMO90]. This means that an object-oriented database is responsible for the storage and retrieval of objects. Relational operations (i.e., relational queries) are performed by methods associated with objects of the type **Relation**. The relational operations are built into the object-oriented database by adding a class **Relation** to it. This thesis follows the latter approach.

The main reason that most people choose OO databases is because they are storing and manipulating data that cannot ordinarily be handled by conventional databases. Good examples of these are CAD projects and CASE tools. These types of applications generally require the ability to store and manipulate graphical objects as opposed to textual objects. In contrast, relational databases serve a very useful purpose in most business applications where the bulk of data being stored and manipulated is simply textual or numeric data that can be stored and manipulated by normal conventional means. The idea behind this thesis is to join the two different types of database paradigms into a single fully functioning database model and to consider alternative ways of implementing relational operations in an object-oriented database.

This relational/object-oriented database can be shown to be a complete relational system because it provides the five core relational operations upon which all other operations can be constructed[1]. The five basic relational operations are built into the database management system (DBMS) instead of creating an artificial layer above the DBMS for them. This allows the user to create a database that is capable of storing and retrieving objects as well as performing SQL-like[2] queries on the database.

The database is designed in Prograph[3] [TGS88a, TGS88b, TGS91], an object-oriented programming language (OOPL) available on the Apple Macintosh[4]. This language was chosen because it contains primitive operations for all database functions (i.e., disk reads and writes, opening keys, tables, etc.), so there is no need

---

1. This will be discussed further in Chapter III.

2. SQL stands for Structured Query Language and was designed and implemented by IBM Research as an interface for a relational database system [EN89]. It has become the defacto standard for the relational database industry.

3. Prograph is a trademark of The Gunakara Sun Systems, Ltd.

4. Apple and Macintosh are registered trademarks of Apple Computers, Inc.

to create them. The database system created as part of this thesis is a functional, albeit minimal, R/OODBMS. The number of 'extra' features was minimized for clarity. Prograph also handles list processing and manipulation of non-conventional objects (i.e., pictures, sounds, etc.) very easily which is important to the project's design.

The remainder of this thesis is organized as follows. Chapter II is a survey of the literature that forms the background for this research. It sets the stage for future discussion in this thesis, and provides an overview of the main topics of this thesis: object-oriented programming, databases, and Prograph. Chapter III presents a detailed description of the problem addressed in this thesis, the implementation of relational operations in an object-oriented database. There is also some discussion on how this idea has been implemented by others, and problems associated with these implementations. Chapter IV describes the findings of this research. Chapter V provides a summary, conclusions, and suggestions for future research. Appendix A is a description of how to use the tools provided to create a database application. Appendix B is the graphical representation of the attributes and methods for each class and Appendix C is the source code for the implementation of this system.

II. SURVEY OF THE LITERATURE

This chapter deals with three major topics: object-oriented programming (OOP), database management systems, and the object-oriented programing language Prograph. Basic terminology and concepts are discussed in this chapter. No assumptions are made about the readers level of knowledge in these three areas. However, some familiarity with OOP and database topics may be helpful in fully understanding the material. This chapter is intended to serve as an introduction to these three topics, laying the groundwork for the rest of this thesis.

A. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is a relatively new area of programming whose origin has been attributed to the programming languages Simula and Smalltalk [Booc91,Mica88, SB86]. Although OOP seems to be the hottest sales item in programming and program design today, there are very few standards that clearly define it. Most agree that for a language to be considered object-oriented it must support at a minimum objects, classes, and inheritance [Nels90a].

Creating complex applications using an object-oriented programming language (OOPL) is usually simpler than designing the same program using a more conventional procedural language. This is because OO design more closely mirrors the real world entities being modeled. Also, the use of encapsulation (data hiding) and inheritance make classes more reusable. As an example, a program written in C in a windowing environment that prints a "Hello World" window takes approximately 42 lines of code. The same program written in Actor (an object-oriented programming environment) takes just two lines of code. This is because there is a Window class declared in the class hierarchy of Actor and the programmer only has to create an instance of this class to make the window. [Wu90]

The development of OO programs is dependent upon the careful design of the classes and their relationships to one another. If the classes accurately reflect the system being designed and if the external interfaces are designed properly, creating a working program is fairly simple. This is the reason that the design of a program is very closely tied to the actual implementation. In Ada, for example, the actual structure of the program may not reflect the structure of the system due to language constraints [Mica88]. Ada has no mechanism to support inheritance, so the design of a class would most likely include some variables and procedures that have already been defined elsewhere. Ada does takes advantage of libraries of different types of functions such as math libraries, etc.; however, the programmer can only use the functions provided in the library 'as is'. If there is some functionality provided by a certain library function, but it does not work exactly the way the programmer wants, or if he would like to add functionality to the existing function, then the programmer must make a copy of the function to be used in his program. This is because there is no way to use an Ada library function as a foundation to be built upon.

The following sections present a more detailed introduction to some of the specific areas of object-oriented programming. The main areas to be discussed are classes and objects, inheritance, encapsulation, reusability, and polymorphism.

1. Classes and Objects

A *class* is defined as "a set of objects that share a common structure and a common behavior" [Booc91; page 93]. An *object* is a set of self-contained variables with a set of procedures which operate on them [Nels90a]. They do not exist in the text of the program, but rather in the memory of the computer while the program is being executed [Meye88]. Another way to view a class is as a framework or a blueprint that describes all instances. It is also helpful to visualize a class as a static entity, and an object as a dynamic entity [Meye88]. That is, once a class is defined

it does not change. In contrast, objects are constantly being created, modified, and deleted during the program's execution. For example, if the structure of an object is defined as having attributes labeled X, Y, and Z, then the values stored in X, Y, and Z can change many times during the life of the object.

It is helpful to think of a class as the general description of an object. For example, one could declare a class called **Person** which describes all the common features of people in general. A specific **Person**, such as John Smith, is an object (instance) of this class. The class description serves as an abstract description of related objects and how they interact with each other and the outside world. All objects of class **Person** share the same structure (attributes) and behavior (things they can do). It is this basic structure and behavior declaration that makes up the class definition. The value of a certain attribute of a **Person** may be different for every instance, but all instances have the same type of attributes.

The description of a class is broken down into attributes and methods (behaviors). Continuing with the **Person** example, some attributes of this class might be *weight, color_hair, birthday,* etc. (see Figure 2.1[1]). Attributes of a class can be further broken down into two distinct groups: class attributes and instance attributes.

Class: Person

    Superclass: none
    Class attributes: total_population
    Instance attributes: weight, color_hair, birthday
    Methods: be_born, do_work

Figure 2.1 Example Class Definition

---

1. This figure is based upon the language independent class definition as presented in [Nels90a].

Class attributes not only share their names with all the instances of the class, but the value of the class attribute is also shared by all instances [Nels90a]. For example, in class **Person**, there might be a class attribute *total_population*. As each new **Person** is created, *total_population* would be incremented, and as an instance is destroyed (dies) *total_population* would be decremented. However, the value is the same for all instances of the class. In other words, if one instance of **Person**, say John Doe, requires the *total_population* for some computation, the value is locally accessible to him and any modification to the value will be immediately visible to all other instances of the class.

Instance attributes share only their names with other instances of the class; the values stored in the instance attributes can (and usually do) differ from one instance to another. The attributes *weight, color_hair,* and *birthday* in Figure 2.1 are examples of instance attributes. While each instance has this set of attributes, the values stored vary from one instance to the next.

Methods describe actions for classes. They are accessed by passing messages to an object. The concept of *message passing* comes from Smalltalk where the means for causing an object to perform some function (method) is through message passing [Booc91, Nels90a]. Methods defined for the class **Person** in Figure 2.1 include *be_born* (make a new instance of the class) and *do_work*. In some languages the only way to access an object's attributes is through the object's methods. This allows the creator of a class to modify its internal structure without the outside users of the class being aware of the change. This idea is called *encapsulation*, and is discussed in Section 3.

The *external interface* of an object is the way the object appears to the users of the class. In C++ terminology, the external interface of an object is the set

of public[2] attributes and public methods, as well as the specifications for these methods. The external interface of an object also describes the methods available for the object and how the messages for the object should be appear. Protected methods are available to a class and its descendants, and although they are not part of the end-user's view of a class, they are visible to descendants of a class and the end-user must be aware of these methods and attributes.

One particularly interesting type of object is the composite object (sometimes referred to as an aggregate object). A *composite object* is an object that is made up of other objects. In other words some (or all) of the attributes of an object are themselves objects. These objects that make up the composite object can be either sub-objects, or dependent objects, the distinction being dependent objects are actually objects, and sub-objects are pointers to objects. [Nels90a]

2. Inheritance

Inheritance is what separates OOPLs from procedural languages such as Ada or Pascal [Mica88]. *Inheritance* can be defined as a mechanism that allows for code sharing. It allows new classes to be defined based on an existing class or classes [Nels90a, SB86]. The existing class is referred to as the *superclass*, while the new class is referred to as a *subclass* of the existing class. In Figure 2.2, class A is the superclass of class B, and class B is a subclass of class A. Class B inherits all of the attributes and methods of class A (including any inherited by A); it may also define new attributes and methods to augment the inherited ones.

There are two types of inheritance, single inheritance and multiple inheritance. *Single inheritance* means that a class can have 'at most' one superclass

---

2. Public methods/attributes are those methods/attributes that are available to all methods and users of the class.
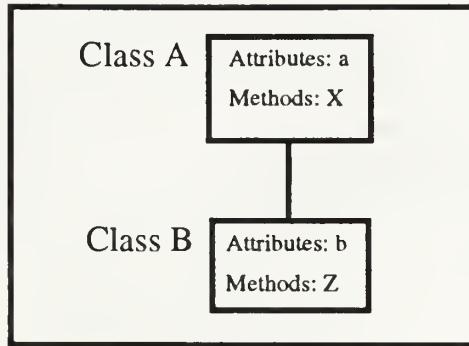
8

Figure 2.2 Superclass/Subclass Example

and is generally referred to as simply 'inheritance'. *Multiple inheritance (MI)* means that a class can have many superclasses.

With inheritance, unfortunately, comes several potential problems. The biggest problem is that of naming conflicts, which is when a class defines a method or attribute using the same name as an inherited one, either by choice, or by accident [Nels88]. Generally if this happens the inherited method/attribute is inaccessible in the new class. The problem becomes much more severe in languages that support multiple inheritance. For example, if there is a method X in class A and a method X in class B, and class C inherits from both A and B (see Figure 2.3), what happens to method X is very much language dependent [Mica88, Nels90a,NMO91]. A similar situation exists with the attribute q.

One possible solution is to have the compiler check for name conflicts and flag them as errors, as in the language Eiffel [Meye88]. Eiffel does not permit a class to inherit a method with the same name from two or more parent classes. Any name conflicts must be resolved through the use of the *redefine* or *rename* commands when declaring the subclass [Meye88]. However, this requires the programmer to have some knowledge of the inner workings of the classes in the inheritance hierarchy, which violates the concept of encapsulation. An alternative notion to this is the *super* construct in languages such as Extended Smalltalk or CommonObjects
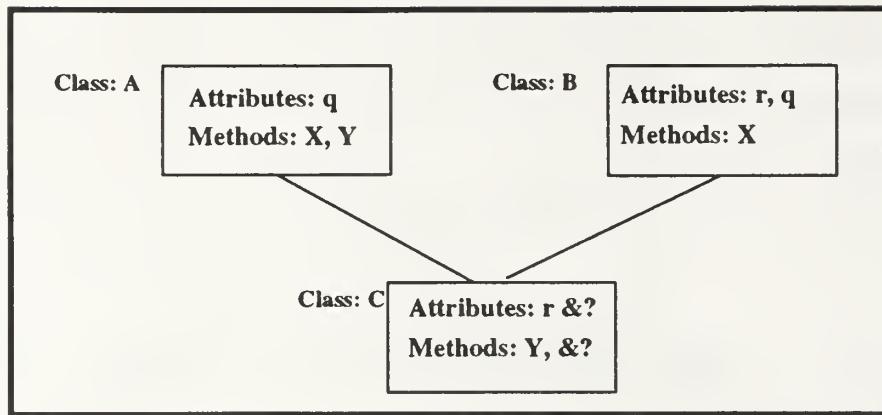
9

Figure 2.3 Multiple Inheritance Problem

[Mica88]. This allows programmers to formally declare which superclass to inherit a method from [Mica88].

A different approach is to create a precedence list to determine which class to inherit from. A *precedence list* is a list that is established, either by the language or by the programmer to determine which classes are 'senior' to other classes. This precedence list can be generated in a number of ways. One way is through a depth-first traversal of the inheritance subgraph starting with the class in question. This is the approach taken in the languages Flavors and Loops. Another way to create the precedence graph is by using a programmer-defined precedence algorithm that resides in the metaclass of the class being defined. This is the approach taken in languages such as CommonLoops. [Mica88]

Inheritance allows the programmer to define objects as they relate to other objects in their world view. Consider the example of an employee database. One of the classes that needs to be defined is **Employee**. This class could have attributes for *name, age, sex, birthday,* and *address* (see Figure 2.4). If the company has salaried and hourly employees, you could create separate classes for these. **Salaried** would have the attributes *salary, years_of_service, job_title*. **Hourly** would have the

attributes *hourly_wage, position*. Since both **Hourly** and **Salaried** are kinds of **Employee**s, they both inherit from the class **Employee**.
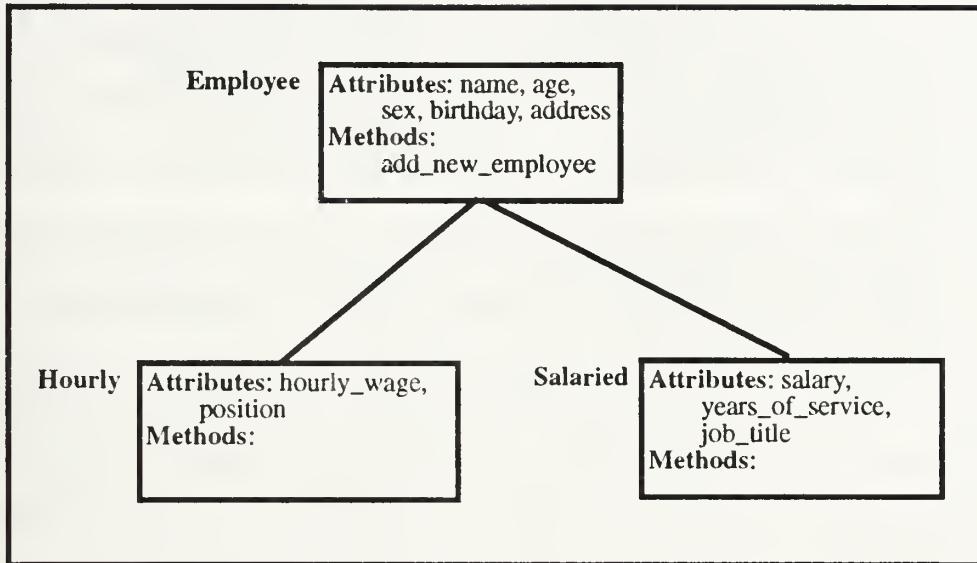

Figure 2.4 Employee Example

Objects of type **Hourly**, for example, would have the inherited attributes *name, age, sex, birthday,* and *address*, along with the locally defined *hourly_wage* and *position* attributes. Objects of type **Salaried** would have the same inherited attributes as **Hourly**, along with the locally defined attributes *salary, years_of_service,* and *job_title*. Thus, the subclasses of **Employee** have access to the attributes defined in the superclass as well as those in their own local definition (see Figure 2.5).

Along with attributes, methods are also inherited by subclasses. An example might be *add_new_employee*, which could be defined in the class **Employee**. If the programmer wants to create a method for making a new hourly employee, he writes the method *add_hourly_employee* in the class **Hourly**. This method could first call *add_new_employee* which would return an instance of class **Employee**, then it would add the specific values for an hourly employee. The call to
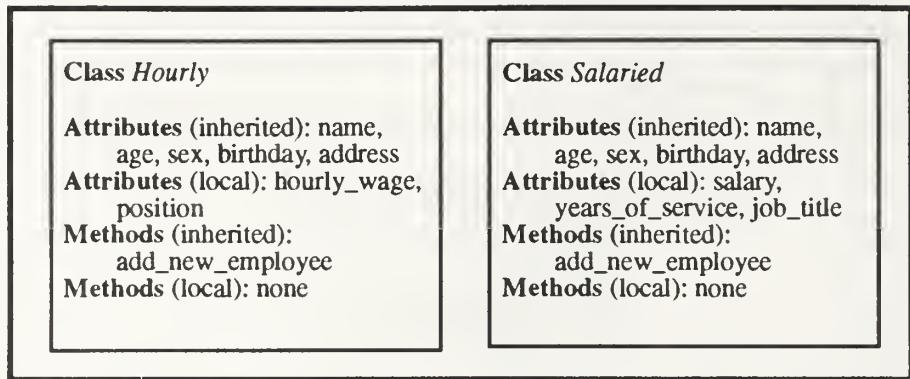
```
┌─────────────────────────────────────────────────────────────────┐
│  ┌──────────────────────────────┐  ┌──────────────────────────┐  │
│  │ Class Hourly                 │  │ Class Salaried           │  │
│  │                              │  │                          │  │
│  │ Attributes (inherited): name,│  │ Attributes (inherited): name, │
│  │    age, sex, birthday, address│ │    age, sex, birthday, address│
│  │ Attributes (local): hourly_wage,│ Attributes (local): salary, │
│  │    position                  │  │    years_of_service, job_title│
│  │ Methods (inherited):         │  │ Methods (inherited):     │  │
│  │    add_new_employee          │  │    add_new_employee      │  │
│  │ Methods (local): none        │  │ Methods (local): none    │  │
│  └──────────────────────────────┘  └──────────────────────────┘  │
└─────────────────────────────────────────────────────────────────┘
```

Figure 2.5 Hourly and Salaried Attributes and Methods

add_new_employee would look inside the class **Hourly** for a method called *add_new_employee*, since it does not exist there it would look to the superclass for a method with that name. Since it is there it would execute; if not found there, then the entire inheritance hierarchy would be searched until the method is found or the top level of the class hierarchy is reached, which would result in an error condition.

3. Encapsulation

Encapsulation can be defined as "the process of hiding all of the details of an object that do not contribute to its essential characteristics" [Booc91; page 46]. It is also referred to as data hiding. In OOP, the user of a class should only need to know how to call a method and what methods are available to manipulate an object. The user does not need to know the internal details about how a specific method works, or what attributes the object has. All the user needs to be aware of is what message is passed to an object, and what will be returned by the object.

Encapsulation is an important concept related to object-oriented programming and modularity of code. If the details of objects are hidden (i.e., they are encapsulated), then program developers can specify what they want and how it should interface; objects can then be built to specification without requiring

information about the exact structure of the object or how its methods are implemented. It also allows for the code to be improved/modified without affecting how end users access the object.

One language that supports the idea of encapsulation is C++. The way to fully encapsulate attributes or methods in C++ is by declaring them as *private*, which means that they are only visible to methods of that class. Thus the implementation details are known only to the class that declared them. The external interface of the class could then be declared as *public* (visible to everyone). Additionally, C++ offers another alternative: declaring attributes/methods as *protected*, which means that they are visible only to the class and its descendants. By limiting the visibility of methods and attributes, encapsulation can be preserved. Most OOPLs provide some means of encapsulation; however, in many cases it is up to the programmer whether or not to utilize encapsulation.

4. Reusability

Reusability refers to "the ability of a system to be reused, in whole or in part, for the construction of new systems" [Mica88, page 13]. This is a very important issue in OOP, as one of the goals of OOP is to reduce the costs associated with software development and maintenance. If programmers develop commonly used structures such as hash tables, sorting algorithms, etc., and then test them thoroughly to prove their correctness, why should they be re-written each time they are required [Mica88]? This should not be necessary if the code exists and can be reused without modification[3]. If the code cannot be reused as is, then perhaps it can be used as a superclass, and the subclass that inherits from it can make modifications or additions to the methods/attributes that are inherited.

---

3. The idea of retesting methods is also addressed in [PK90]. It is pointed out that even though a method may be thoroughly tested in its original context, retesting is required to meet the standards of adequate testing in the new context in which it appears.

Inheritance then becomes a key factor in determining how reusable a system is, because without inheritance the designers of software must either modify an existing class to do what is desired, or copy it and then modify the copy. Both of these alternatives are, however, unacceptable. The problem with modifying the original is the loss of abstraction. The original class may become unrecognizable as each programmer makes his changes. It may also become unusable to some clients because the details that were modified were ones that they relied upon. The problem with making a copy of the original class and then modifying it is the same type of problem associated with maintaining multiple copies of data. Each time the original class needs to be updated for bug fixes or enhancements, all copies of the class also need to be updated. Inheritance allows the original implementation of the class to stand as is, and clients can then use it or inherit from it and modify attributes or methods as required.

Of course, with every benefit there are also drawbacks or side-effects associated with it. The side-effect associated with inheritance as it relates to reusability, is that if a class is properly encapsulated, then the subclass cannot modify the existing attributes or methods without violating that encapsulation. If, on the other hand, the subclass is created to simply take advantage of all (or some) features of the superclass and add some new functionality, then there is no problem with encapsulation.

5. Polymorphism

The ability to have more than one class with methods of the same name but with different implementations is known as *polymorphism* [Nels90a]. "Polymorphism is an important feature of all object-oriented programming languages that allows the definition of flexible software elements amenable to extension and reuse" [Mica88, page 31]. Examples of polymorphism (also called

operator overloading) that are common in most conventional languages are the arithmetic operators +,-,*, and /. If you declare two integers and two real numbers, and then perform addition on each pair, you have used the symbol '+' to perform two different functions; one is integer addition and the other is floating point addition.

Simple polymorphism refers to the ability of different classes to implement the same operation, differently. *Multiple polymorphism* is when a single class can have multiple operations with the same name, but possibly a different parameter list. When a message is sent to a class with multiple methods with the same name, the method used is determined by the parameters passed in the message. [Nels90a]

Polymorphism allows programmers to add multiple methods to classes that share some commonality and therefore use the same name to denote the specific function. This is preferred over creating an artificial name to distinguish two or more methods that are essentially the same with the exception of their parameters, or the receivers of the message. An excellent example is a **print** method for a screen object and a printer object. If the receiver of the print message is a screen object the information will be displayed upon the screen, if it is a printer object the information to be printed will be sent to the printer. There is no distinction between the messages to perform a screen print and printer print, however the result of the method is very different.

B.   DATABASE MANAGEMENT SYSTEMS

Of the three major topics covered in this chapter, databases and database technology is the oldest. It is also a major part of the foundation upon which this thesis is built. This section serves as an overview of database topics and the distinctions between the different types of database technology available in the marketplace today. It is not intended to be a tutorial for novices to learn database systems. The read-

er may refer to "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe [ER89] for a more detailed introduction to this subject.

1. Basic Terminology

A *database* is a logical collection of related data that has some intrinsic meaning [EN89]. It can also be defined as data that is permanently[4] stored in a computer [Nels90b]. These definitions lead to the conclusion that a random collection of data is not a database, which is an accurate assessment. A database is also said to represent a subset of the real world, sometimes called a *miniworld* [EN89].

The tool used to build, store, and manipulate the data stored in a database is the *database management system (DBMS)*. A DBMS can be thought of as a general-purpose software system [EN89] to perform the previously mentioned tasks. Each type of DBMS (hierarchical, network, relational, and object-oriented)[5] has its own particular way to define the data being stored; this is sometimes referred to as its *data-definition language (DDL)*. The way a DBMS stores and retrieves data from the storage medium (disks, tapes, etc.) is through its *storage definition language* and the way a database user sees the data presented is controlled by the *view definition language*. Some DBMSs have these languages separated into distinct languages while others have a single all-purpose database programming language.

Another important concept in database technology is the data model. A *data model* is a way to abstract the data being stored in a database to provide a clearer visualization of the data than computer storage concepts provide. That is, data

---

4. That is, the data will exist after normal termination of the program. The data is "permanent" until the user issues a delete command to remove the data from the database.

5. Only relational and object-oriented approaches are discussed in detail in this thesis.

models represent the data being stored in a form that is more meaningful to the user than the actual form the data takes on the storage medium. For example, consider a data model that represents the data in a tabular form. This would be clearer to most people than a B-Tree data structure that might actually hold the data.

Many different types of data models exist to represent the different types of databases. One model that crosses the boundaries of different database types is the *entity-relationship (ER)* model [EN89]. This model can be used to design a relational database as easily as a hierarchical, network, or object-oriented database. The basic elements of the ER model are entities, attributes, and the relationships between entities. An *entity* is defined as a "thing" in the real world, either tangible (e.g., an employee) or intangible (e.g., a project) and each entity has *attributes* that describe it. For example, an employee object might have attributes representing name, dept, social security number, etc. A *relationship* between two entities means that they are linked by some attribute in each entity. Relationships can represent one-to-one, one-to-many, or many-to-many associations between relations. An example is a company database that maintains employee entities and project entities. A typical relationship might be a 'works-on' relationship where employees work-on a project. An attribute of an employee entity might be **project number** and an attribute of project might be **number**. The works-on relationship would consist of tuples representing every employee/project pair. Quite often the relationship is stored in the database as a table as well because they have attributes associated with instances of the relationship, or because the relationship is one-to-many, or many-to-many.

2. Relational Databases

The relational data model was introduced in 1970 by Dr. E. F. Codd [Codd70]. Since its introduction there has been a great deal of interest in, and

development of, relational database systems. This can be attributed to the ease of use associated with it as well as its firm mathematical foundation. It is also a more abstract way to present a database than hierarchical or network databases, which are closely tied to the physical data structures.

A relational database is, as the name implies, a collection of relations. *Relations* can be thought of quite simply as tables containing entries for each entity being stored [Alag86]. These tables consist of many instances (or rows) of tuples, and a *tuple* (often called a record) is a collection of related data that describes one entity of an ER diagram, or one object being stored. The columns of a relation represent the attributes associated with an entity in the ER diagram.

The key to understanding relational databases lies in understanding set theory from mathematics. The concept of storing data in tables is fairly easy for the average individual to comprehend. Figure 2.6[6] is an example of a very simple relational database displayed in tabular form. One would have to agree that there is little difficulty in determining what is being stored in this database. However, understanding how to obtain specific records or values from the database requires that the user understand the relational algebra associated with performing queries. A *query*, as the name implies, is how a database is interrogated or probed for information. Queries are performed to retrieve specific information from the database. For example, to obtain a list of employee names whose SupSSN is equal to 222333444 requires two select operations and a project operation. These relational operations, plus union, set difference, and Cartesian product, are the subject of the following section.

---

6. The name of the primary key is underlined.

Employee

Name	SSN	Dept
Smith	123456789	Sales
Borg	222333444	Mktg
Jones	256789043	Sales
Williams	456782910	Sales
Edwards	598320982	Mktg

Department

dept-name	Location	SupSSN
Mktg	New York	222333444
Sales	Los Angeles	256789043

Figure 2.6 Simple Relational Database

a. Relational Algebra

One of the most commonly used relational operations is the *select* operation. This operation is used to retrieve entire tuples from a relation. The select operation gets the tuples based on a selection condition. The *selection condition* is a boolean expression made up of one or more expressions of the following form:

<attribute name> <comparison operator> <constant value>; or

<attribute name> <comparison operator> <attribute name>

where <attribute name> represents an attribute name in the relation being operated on, <comparison operator> is one of the operations in the set ($<$, $>$, $=$, $\leq$, $\geq$, $\neq$), and <constant value> represents a constant value. There can be any number of these clauses joined by the Boolean operators AND, OR, or NOT. [EN89]

The *project* operation is similar to the select operation except that where the select operation retrieves specific rows from a relation, the project operation retrieves entire columns from a relation. Also, instead of a selection

condition to determine which tuples to chose, the project operation simply lists the attribute names or the columns to be retrieved. [EN89]

The next three relational operations can be classified as *Set Theoretic Operations*. That is, they are operations that are performed on entire sets (relations). The basis and proof of correctness of these operations is directly related to mathematical set theory proofs and concepts. [EN89]

The *union* operation creates, from two union compatible relations, a new relation that contains every tuple from both of the original relations with duplicate tuples removed, just like the union of two sets in mathematics. *Union compatible* means that both relations have the same number of attributes and the attributes are of the same domain (e.g., the attribute pairs are both from the set of 9 digit integers). Figure 2.7 shows two union compatible relations, the Employee and Supervisor relations, and Figure 2.8 is an example of the union of those relations.

Employee

EName	ESSN
Smith	123456789
Borg	222333444
Jones	256789043
Williams	456782910
Edwards	598320982

Supervisor

SName	SSSN
Borg	222333444
Stone	567811543

Figure 2.7 Two Union Compatible Relations

The *set difference* operation is defined as the relation containing all the tuples in the first relation but not in the second, as in Figure 2.9. It is denoted by a minus sign ("-") such as Employee - Supervisor.

EName	ESSN
Smith	123456789
Borg	222333444
Jones	256789043
Williams	456782910
Stone	567811543
Edwards	598320982

Figure 2.8 Result of Employee Union Supervisors

EName	ESSN
Smith	123456789
Jones	256789043
Williams	456782910
Edwards	598320982

Figure 2.9 Result of Employee - Supervisor

The *Cartesian product* operation combines tuples from each relation in such a way that each tuple of the first relation has each instance of a tuple from the second relation appended to it. The resulting relation has as many attributes (columns) as both relations combined and the number of tuples is equal to the number of tuples in the first relation multiplied by the number of tuples in the second relation. The Cartesian product of the relations Employee and Department is shown in Figure 2.10.

As can be seen in Figure 2.10, there is quite a bit of redundancy in the resulting tuples as each tuple in the first relation is repeated once for each tuple in the second relation and vice versa. Therefore, the result of a Cartesian product is

Name	SSN	Dept	dept-name	Location	SupSSN
Smith	123456789	Sales	Mktg	New York	222333444
Smith	123456789	Sales	Sales	Los Angeles	256789043
Borg	222333444	Mktg	Mktg	New York	222333444
Borg	222333444	Mktg	Sales	Los Angeles	256789043
Jones	256789043	Sales	Mktg	New York	222333444
Jones	256789043	Sales	Sales	Los Angeles	256789043
Williams	456782910	Sales	Mktg	New York	222333444
Williams	456782910	Sales	Sales	Los Angeles	256789043
Edwards	598320982	Mktg	Mktg	New York	222333444
Edwards	598320982	Mktg	Sales	Los Angeles	256789043

Figure 2.10 Result of Cartesian Product of Employee with Department

usually used only for a specific purpose and then discarded due to this excessive redundancy.

The five relational operations just covered have been demonstrated to be a complete set of relational algebra operations [EN89]. That is, it has been shown that any other relational operation can be constructed from some combination of these five basic operations. Thus any relational database system can be considered complete if these five operations are included[7].

b. *Limitations of Relational Databases*

Probably the most obvious and important limitation of conventional databases (i.e., relational as well as hierarchical and network) is the lack of support for large unstructured data such as sounds or pictures. As Kim points out in [Kim91], conventional databases have served us well in the application domain they were

---

7. Of course, efficiency of operations is another issue. For instance, it may be much more efficient to implement other operations, such as intersection, directly rather than in terms of these five basic operations. [Nels88]

originally developed for, namely business and payroll type applications. However, many of today's applications, including CASE tools and CAD programs, require a more dynamic storage and retrieval system such as the object-oriented data model.

Additionally, relational databases tend to loose the structure of the data being modeled as it is normalized. To efficiently model data in the relational model it must be *normalized*, or flattened out to remove redundancies and dependencies. In doing so the original structure of the data is often lost. This causes the data abstraction to be lost and removes much of the original meaning of the entity being stored.

3. Object-Oriented Databases

In recent years there have been quite a few object-oriented database management systems (OODBMS) developed. This section briefly describes some features common in most OODBMS as well as discussing various problems and limitations associated with them. Two example OODBMSs to be discussed herein are Gemstone [Serv89a, Serv89b] and Vbase [Onto88] (the predecessor of Ontos [Onto90]).

What differentiates an OOPL from an OODBMS is persistence of objects. An OODBMS allows the user to create and manipulate objects (as does an OOPL), but it also provides for permanent storage of the object so it can be used again in another session. Most OOPLs do not have the facilities to save objects from one session to another[8]. Aside from this difference most OODBMSs are very similar to OOPLs.

OODBMSs allow users to declare classes and create instances of these classes just as OOPLs do. However, when the user requires an object that has been

---

8. It should be noted that Prograph is one of the few OOPLs that does support object persistence without requiring a database file to hold the objects. This feature will not be used for the purposes of this thesis, a separate database file will be created instead.

written to secondary storage, the OODBMS retrieves the object from the storage medium and puts it into the correct form so that the user can manipulate it. An OOPL can only retrieve objects stored in memory during a single session. The OODBMS handles the reading and writing of the objects to and from the storage medium in a way that is transparent to the user, just like conventional DBMSs. However, OODBMSs also maintain the methods associated with the class of the object being stored so they retain their ability to function and perform operations. Conventional databases store only data values and therefore they can only be used by a separate program; objects stored in an OODBMS, on the other hand, are able to function when they are retrieved from the database.

All OODBMSs surveyed have a database programming language associated with them as well as a class library. There are as many languages as there are systems. Some languages are variants of Smalltalk-80 (e.g., Opal, the Gemstone data definition and manipulation language), others are variants of C or CLU (e.g., COP, the data manipulation language of Vbase). There are also many different class libraries which contain class definitions for such classes as **Collection**, **Set**, **Dictionary**, **Association**, and others. These classes provide a means to organize the objects being stored in the database. Without these classes an OODBMS would not be very usable, it would merely store all objects in one location without a logical connection and, as discussed earlier, this defies the definition of a database.

When creating objects, the user will generally chose to store each object in a collection of some kind based upon the type of data/database being modeled. If, for example, the user wishes to model a relation, an instance (or descendant) of **Set** might be created, because relations are defined as sets of related tuples. By storing objects within a descendant of **Collection** the data can be thought of as one logical unit and manipulated as such. Depending upon the subclass of collection being used, the data being stored can also inherit qualities that provide easier (or faster) access

24

to each object. For example, if a collection of objects is related in such a way that the objects are all different and each has a key/value associated with it, an instance of **Dictionary** might be created to hold the objects. [FN92a]

OODBMSs also provide many of the features expected in conventional database systems, such as access control and concurrency control. They generally provide methods for locking individual records/objects during updates, just like conventional databases. Since the OODBMSs surveyed are designed to run on multi-processing architectures, concurrency control is built into the database kernel or monitor running on the host computer.

It has been said that relational databases are value-based whereas object-oriented databases are identity-based [ZM90]. This is a result of the previously discussed idea that object-oriented databases capture the function as well as the value of each object.

4. Relational/Object-Oriented Databases

Although there has been relatively little work done in this area of database research, it is probably one of the more important areas. This is because there exists a very large base of relational database applications, and SQL-based languages have become the defacto industry standard [Alag86]. While most agree that object-oriented databases are the way of the future, many hours and dollars have been invested in relational systems and people are generally unwilling to forsake relational databases in favor of a new model. What the relational/object-oriented model promises is a way to have all the benefits of the newer OO approach without abandoning the "tried and true" relational approach.

It has already been established that a major failing of conventional relational databases is the inability to represent complex data types. This is very important if technology is to continue to move forward. A relational/object-oriented

database system would take the best of both types of database technology and combine them into a single more powerful database tool.

It has been proposed that the capability to perform relational operations could be added to any object-oriented system [Nels88, NMO90]. A prototype system, called a relational object-oriented database management system (ROOMS), provides an interface that could be used as a stand alone database system or added to any existing OODBMS. The system is designed around a **Relation** class, a **Record** class, and a **Database** class. The workhorse is the **Relation** class. It includes of methods for each of the five basic relational algebra operations as well as methods to display a relation and add/delete records from the relation. This approach will be discussed in more detail in the next chapter.

C. PROGRAPH

Prograph [TGS88a, TGS88b, TGS91] is billed as a "Very high-level pictorial object-oriented programming environment" for the Apple Macintosh. This means that the level of abstraction from machine language is about as distant as one can get (at this time, anyway). For example, programs written in assembly language are directly tied to a specific processor, and the actual instructions have a nearly one-to-one mapping with the machine instructions for that specific processor. In contrast, Prograph allows the programmer to design and implement programs as a number of objects which interact with one another to produce desired results. This is common to all OOPLs; however, Prograph allows the programmer's mental image to be transferred to the computer in an iconic fashion, thereby reducing loss in translation to textual form. To aid the programmer in seeing the application in terms of classes, methods, and attributes, Prograph makes use of both icons and the Macintosh operating system's windowing environment. Programs are built as dataflow diagrams representing the data that flows through the program. This is in contrast to traditional

programming languages which treat data as something stored away in memory somewhere, and only handled as necessary.

Prograph is a hybrid OOP language. This is because Prograph supports primitive language types such as character, integer, boolean, etc. A pure OOP language has no primitive language types; everything is an object [Booc91, Mica88]. Prograph, like C++, has primitive data types as part of the language that are not in the inheritance hierarchy. Another feature that makes Prograph a hybrid language is the concept of universal methods. These are methods that do not belong to any particular class, but can be called from any method in any class [TGS88b]. Support for universal methods frees the programmer from having to create specialized classes somewhere in the inheritance hierarchy to perform one specific task.

1. The Language

Since Prograph is a dataflow language, data is active, not static. That is, data moves through the program rather than sitting in a memory location waiting to be processed as in the von Neumann model. In a dataflow language, as soon as data is available to each input of an instruction, it can execute. This represents a key difference between dataflow languages and sequential text-based languages, in that the execution of a von Neumann machine is based on the process of fetching an instruction, executing it, and then fetching the next instruction, and so on. This provides a single thread of control from instruction to instruction. In contrast, the dataflow machine's instruction cycle detects when all required inputs are active (i.e., have data available), fetches the instruction, executes it, and generates the data to be output by the instruction. Since there are many possible flows of control in a Prograph program, it supports concurrent processing [TGS88a]. Obviously, since the programs used in this thesis are intended to be run on a uniprocessor system (the Apple Macintosh), true concurrency is not possible.

27

Some additional Prograph terminology is necessary to understand the examples used herein: terminals, roots, and primitives. *Terminals* are the input to methods and primitives, and *roots* are the outputs. *Primitives* are the procedures (or functions) that are built into the language.

An example of a primitive is get-file. This primitive accepts as input (via its single terminal) a list of file types to be displayed in a standard Macintosh open file dialog box (see Figure 2.11). The outputs of this operation are: the name of the file selected from the scrolling list, the volume identifier, and the file type. This is just one example of the power of primitives. They encapsulate much of the detail involved in creating certain displays, as well as performing such functions as arithmetic, list processing, and so on.
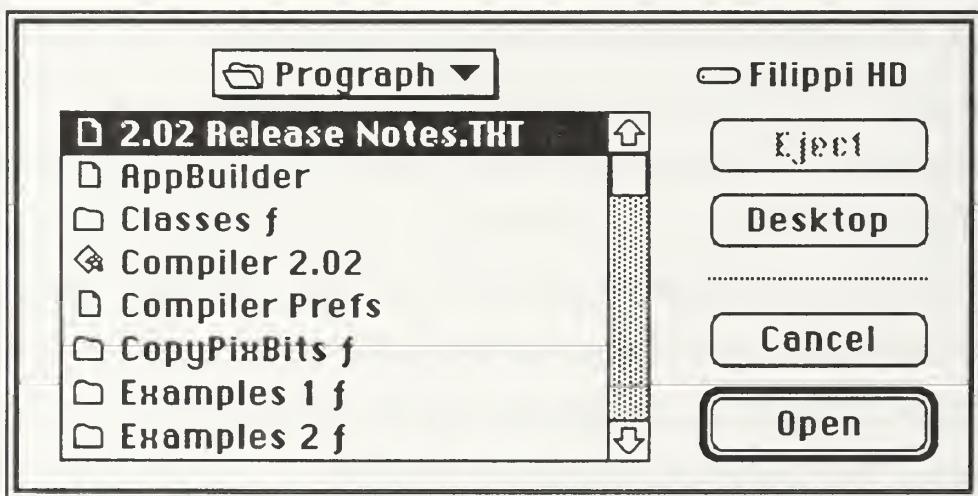
Figure 2.11 Result of get-file primitive

Prograph is one of the few OOPLs that supports persistent objects. *Persistents* are defined as data or objects that exists from one execution of a program to another. They are created and displayed in a Persistents window that is separate

from the Classes window and the Universal window. Persistents are created in the same way as a class or method, and can be double-clicked to display their values. Persistents allow the user to manipulate objects and store them within the program so that they can be used later during the execution of the program, or recalled during another execution of the program.

a. Classes

Prograph classes are represented by hexagonal icons displayed in the Classes window (see Figure 2.12[9]). Within the window are the class hierarchies for the program. There can be as many separate hierarchies as the application requires. This is different from some other languages, such as Smalltalk, which allow only a single class hierarchy. In Figure 2.12 there are three distinct hierarchies. Two of them contain only one class each and the third has as its root the class **Animal**. The lines between classes in the Animals hierarchy represent the inheritance links between various classes.

The class icon itself represents the component parts of a class, its attributes, and its methods. The left-half of the icon represents the attributes of the class, the right-half represents the methods. Double-clicking on the left half opens the attributes window for the particular class. Similarly, double-clicking the right half opens the methods window for the class.

To create a new class the programmer points to white space in the *Classes* window and clicks the mouse button. Once the icon appears the programmer gives the class a name and defines its attributes and methods. If the name selected by the programmer already exists, then the system will not accept it, warning the user that the chosen name is already in use.

---

9. This figure as well as those that follow in this section, are taken from an example in Part II, Chapter Six of the Prograph Tutorials [TGS88a].
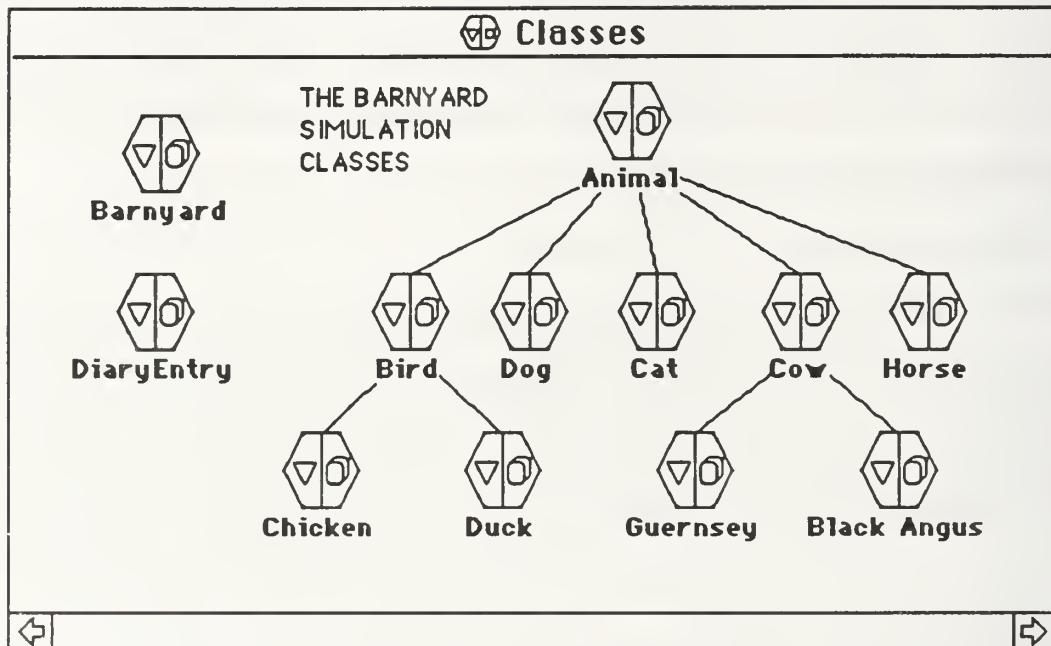
Figure 2.12 Barnyard Simulator Classes Window

b. *Attributes*

Attributes are displayed in the attributes window. Class attributes are represented by the hexagon shaped icons, and instance attributes are represented by inverted triangles. The attributes window is divided into two parts by a horizontal line. Above the line are class attributes, below it are instance attributes. The attribute windows for the classes **Animal** and **Guernsey** are shown in Figure 2.13. The class **Animal** has no class attributes, only instance attributes which are defined locally. We know this because the icons for the attributes do not contain a downward pointing arrow. In contrast, the attribute window for **Guernsey** has one class attribute, **herdMembers** which appears above the horizontal line, and instance attributes **name, age,** and **food**, below the line. The three instance attributes are all
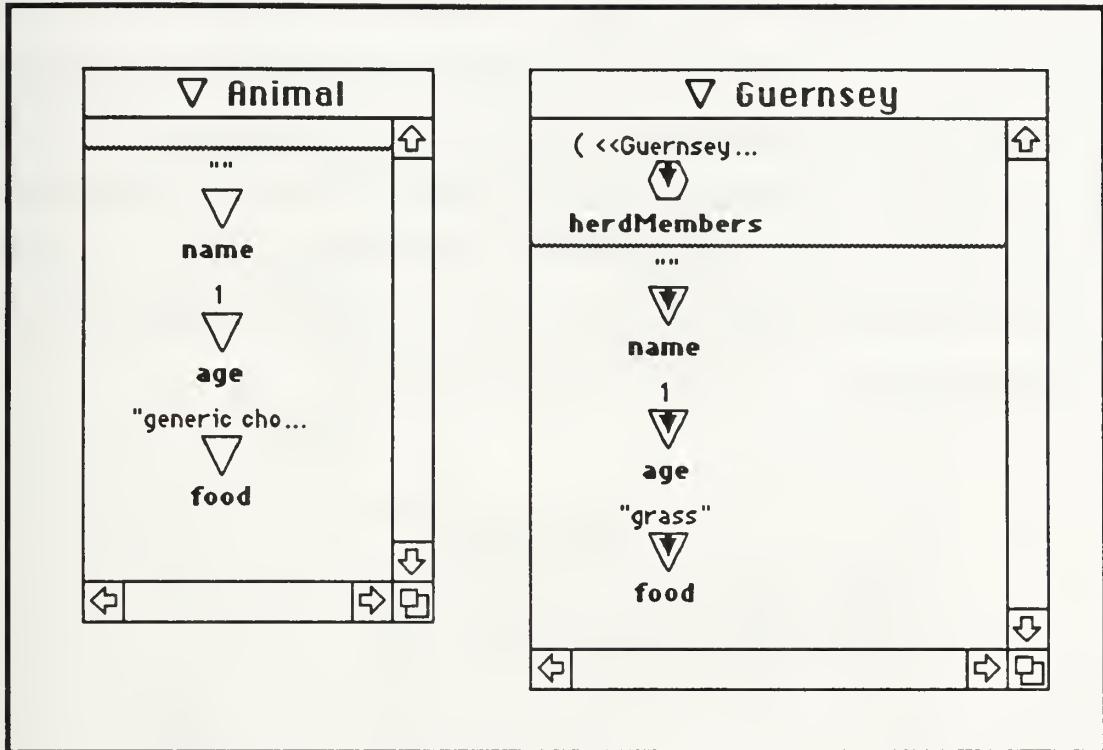
Figure 2.13 Sample Attribute Windows

inherited via the superclass **Cow**[10]. This is shown by the downward arrow in each instance attribute icon (inverted triangle icon). If there were any attributes particular to only a **Guernsey,** they would appear below the inherited attributes in the window. The class attribute is also inherited from **Cow** so it has the downward arrow in the icon (the hexagonal icon).

Attributes can be assigned initial values by double-clicking on the icon and changing the value in the attribute editor. Attributes can also be more than simple data types, they can be instances of other classes; this is how you represent a composite object in Prograph.

---

10. Any attributes inherited by Cow are also passed on to its subclasses.

c. *Methods*

Methods are represented by an icon that contains a mini-dataflow diagram (see Figure 2.14). The hexagon shaped icon labeled <<>> is a special kind of method called an initialization method or *instance generator*. This instance generator method is invoked whenever an instance of that class is created. It allows you to tailor creations of an instance. This method also overshadows the instance primitive (*overshadowing* is the term used in Prograph for redefining inherited attributes/methods).
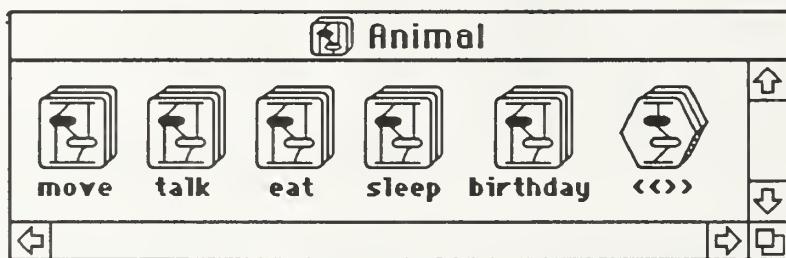
Figure 2.14 Class methods window

When a method is selected and opened (by double-clicking the icon), a case window such as the one in Figure 2.15 is opened. This is where the dataflow aspect of the Prograph language is most apparent. In this example, there are three basic types of operators. The operators with the concave left side labeled **food**, **name**, and **what happened** are **Get** operators. These retrieve the values of the attributes that match their label. Operators with a convex left-side (such as the bottom occurrence of **what happened**) are **Set** operators. These set the values of the attributes that match their label. The other types of operators shown are primitives. For instance, the two labeled **join** catenate two or more strings to produce a single string. The long narrow bars at the top and bottom of the case window are the input

bar and the output bar. These are where the data flowing in and out of the method are attached. In this example, there are two inputs (an instance of the class **Animal**, and the current eventRecord) to the method, but no outputs.
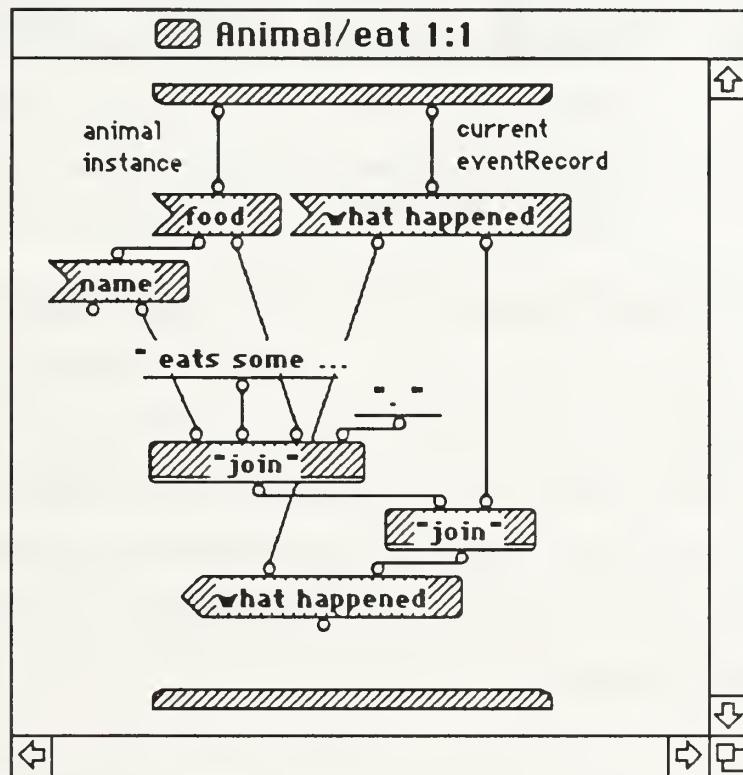
Figure 2.15 Case Window for Animal/eat

d. *Message Passing*

Message passing in Prograph is accomplished by creating a simple operation with the name of the method being called. Once the method has been named and the Enter key (Return key) has been pressed, Prograph assigns the operation the correct arity based upon the arity of the method called. *Arity* refers to the correct number of terminals and roots for the operation. Figure 2.16 shows the

five ways to reference a method. They are, respectively: universal reference, explicit reference, data-determined, context-determined reference, and super.
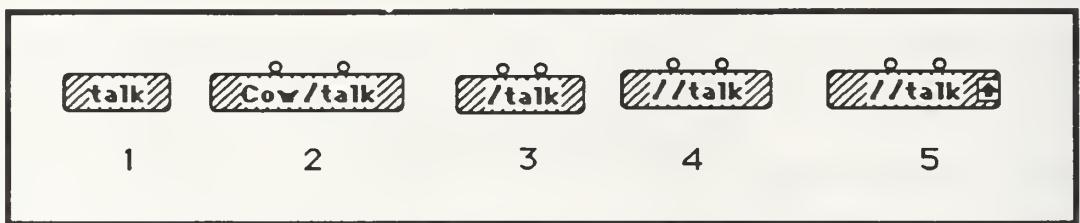
Figure 2.16 Method References

A *universal reference* is when the method being called resembles Example 1 of Figure 2.16; that is, there is no slash ("/") preceding the name of the method. The method to be executed is located in the Universal methods window. Universal methods are similar to global functions in other programming languages, because they have a global scope. In an object-oriented language such as C++ these types of functions could be stored in a Global dictionary so they could be used by any method that needs them.

An *explicit reference* is always in the form **ClassName/MethodName** (see Example 2 of Figure 2.16). If the method is not found in the specified class, its ancestors are examined for the appropriate method name, but the search begins with the class named before the slash. These types of methods are not directly related to any object-oriented concept. They are provided mainly as a way of supporting traditional programming concepts.

A *data-determined reference* is similar to the OO concept of message passing and is denoted by a method name preceded by a slash (see Example 3 of Figure 2.16). The method called is based upon the instance of the data arriving on the left-most terminal of the operation. In other words, the class to search for the

named method is the same class as the instance arriving on the left-most terminal. Data-determined references are examples of single polymorphism.

A *context-determined reference* is akin to the OO concept of sending a message to one's self and is denoted by the method name preceded by two slashes (see Example 4 of Figure 2.16). The class to be examined for the named method is the same class the calling operation exists in. For example, a class **Cow** might contain a method *new_cow* that creates a new instance of the class **Cow** and names the new instance by calling the method *name* in class **Cow**. The operation in *new_cow* that names the new cow is labelled *//name*, this tells Prograph to look for *name* within this particular class.

The fifth type of operation is called *super*. It is a context-determined reference that searches for the appropriate method, not in the class it is called from, but its superclass. A super operation is denoted by an up-arrow in the right side of the operation icon (see Example 5 of Figure 2.16). The super operations allow a method to use a parent's method and then add its own functionality to the parent's method.

e.   *Control Structures*

Even though Prograph is a dataflow language, flow control can be imposed and is in fact often necessary to obtain desired results. The primary way to affect the program flow is through the use of controls. *Controls* are attached to certain operations and are activated based upon success or failure of the operation. The default setting for all operations is to activate on success. This default is changed by selecting the operation desired and selecting the appropriate control from the Controls menu.

Controls are represented by a small square icon attached to the right side of an operation. Within the square is a check mark (√), indicating activate on

success, or an X, indicating activate on failure. There are also other icons within the small square indicating what action to take if control is activated (e.g., go to next case, continue, terminate, finish, or fail). Probably the most common control is the 'on failure go to next case'. This allows the programmer to execute different operations within a single method based upon success or failure of a test. This could be used as an if-then-else structure or a case structure. An example of an if-then-else structure is shown in Figure 2.17. The ≤ operation has an 'on success go to' (√) control attached to it so if the age of the cow is ≤ 3, control goes to the next case (i.e., Cow/name 2:2) and "Lil'" is appended to the front of the cow's name. If the age ≤ 3 operation fails, control remains in the current case and only the name of a cow is output. The case windows are distinguishable by their title bars. Each has the same method name, but they also have 1:2 or 2:2 to distinguish the first case window from the second. A comparable C code fragment is shown in Figure 2.18.
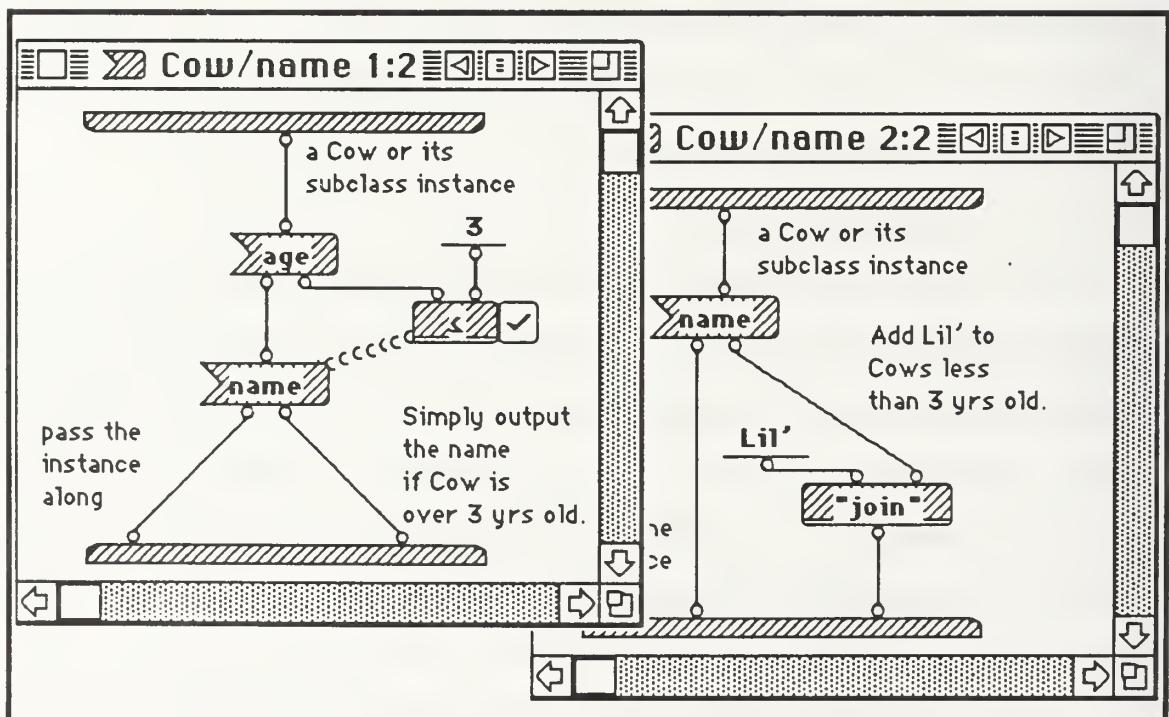
Figure 2.17 Control Structure Example

```
if (cow.age > 3) printf("%c",cow.name);
else printf("Lil' %c",cow.name);
```

Figure 2.18 Example C Code

Another form of flow control is the synchro link. It forces one operation to be executed before the second can execute. This allows the programmer to ensure that two things which could be executed in either order, will only be executed in the desired sequence. In Figure 2.19, for example, there is a synchro link from **show** to **ask**. This is to ensure that the show operation executes before the ask operation.
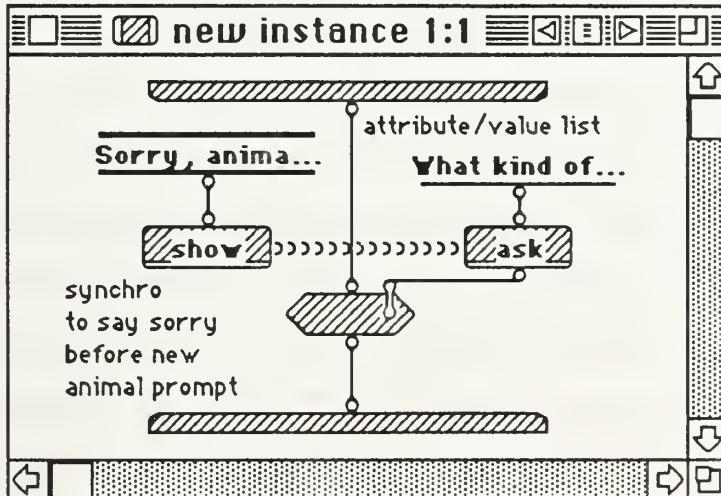
Figure 2.19 Synchro Link Example

Synchro links are a useful tool in controlling the execution of certain operations. However, they do not guarantee that the second method will follow immediately after the first; rather, it only guarantees that the first operation will execute before the second.

37

2. Prograph Database Engine

Before discussing the Prograph database engine, it is appropriate to first define some new terms. The *Database Engine* is the set of primitives which perform all of the functions normally associated with a database management system, including all the functions necessary to create and maintain a disk-based database. These primitives allow the programmer to manipulate databases, the tables stored in the databases, and the clusters that make up the tables. The programmer can also access the keys for a database table through primitives. *Tables* in Prograph databases are composed of clusters which should logically be of the same type (i.e., all the clusters should be an instance of a single class), but are not limited to this convention. In its simplest form, a table represents data that the programmer has grouped together. A *Cluster* represents the individual record being stored in the database. The cluster is not restricted to any specific data types. "A cluster can contain textual, numeric, and boolean (TRUE or FALSE) data, as well as Macintosh structures (such as PICTs or ICONs) and Prograph objects (instances of classes)" [TGS91, page 48]. A cluster is a way of taking an arbitrary collection of Prograph objects and packing them into a single stream of bytes. The breaking up of objects into clusters is transparent to the user; the user always sees the cluster as a collection of data.The database engine allows the programmer the flexibility to model flat-file, relational, object-oriented, and other databases as the programmer deems necessary. [TGS 91]

A database can be accessed in one of three modes: query, update, or shared mode; with the mode being determined when the database file is opened. *Query* mode allows read-only access to a database by multiple users. *Update* mode is for single-user reads and writes to a database. The *share* mode provides multiple users

38

both read and write access to the database.[11] In the multiple user modes, all users must open the database in the same mode (i.e., all users must be in either query mode, or they must all be in share mode, but not a mixture of query and share).

Access to the data stored in the database can be through one of three access methods:[12] direct cluster access, sequential key access, and random key access. **Direct cluster access** means retrieving a cluster based upon its ID (each cluster has a unique ID which corresponds to the byte offset of the cluster in the data file). **Sequential key access** is the way clusters are retrieved based upon the ordering imposed by the value of the key. For example, if the clusters were employee entries and the key was last name, sequential key access would allow retrieval of employees in alphabetical order. **Random key access** is how clusters are retrieved based upon key/value matches. Suppose the employee record desired has the last name equal to Smith; random key access allows the cluster corresponding to the key value equal to Smith to be retrieved without having to visit all the intervening records first.

Prograph databases can contain multiple tables, and each table can contain multiple clusters and multiple keys. In order to manipulate a cluster (record) in a Prograph database, the programmer must first open the database file, then the tables, and finally the keys (if applicable). This hierarchy must be adhered to or the primitives associated with storing and retrieving clusters will fail and return an error code that must be handled by the application.

All database primitives (i.e., database, table, key, and cluster primitives) return as the left-most root an error code. If the error is a zero (0), the operation was successful; if it is any other integer, then there was an error. This allows the program to recover from errors by testing this output and branching as appropriate. It also

---

11. Multiple users refers to the ability to share a database across a network of Macintosh computers. The database would reside on a File Server and the clients could access the data concurrently.

12. These are access methods in the classic database sense, not as in object-oriented methods.

allows the program to close all open database files in the event of an unrecoverable error such as a "Disk Full" error.

III. WHY AN R/OODBMS

The previous chapters were presented to give the reader a basic understanding of OOP, Prograph, relational databases, and object-oriented databases, and to act as a common point of reference for ideas presented in this chapter and the next. This chapter presents a discussion of some of the deficiencies associated in current database systems. There is also a discussion of why a relational/object-oriented approach is advocated and why this approach is taken. Finally, the reasons for selecting Prograph as the language to implement these ideas are covered.

A. DEFICIENCIES IN CURRENT DATABASE SYSTEMS

This section covers the problems associated with building complex database applications using existing database systems. The desired properties of each type of system are also presented.

1. Relational Databases

Relational database technology and its limitations are the driving force behind this thesis. Current implementations of the relational data model are very limited in their ability to handle complex data types. That is not to say that the relational model is no longer of value, as it still serves the purposes of many applications quite well. As discussed in the previous chapter, however, the relational databases that are currently available do not allow users to maintain the desired level of abstraction and encapsulation of the data being stored and manipulated.

a. Limited Number of Data Types

The number of data types available is typically fixed in RDBMSs, and is one area of concern for developers of new types of database applications. Specifically, the lack of support for complex data types such as digitized pictures, digitized sound, and composite objects. Some relational databases currently

available have the capability to store pictures in a database, but not sounds or composite objects, and there is no way to perform a query based on a complex object. In a CAD application, it is desirable not only to be able to store graphical objects, but also to be able to perform queries based upon them. The ability to display graphical objects based upon their properties is also very desirable.

All relational databases include support for simple data types (i.e., strings, numerical, boolean, etc.), and some even provide support for digitized pictures. However, the real world we wish to model is much more complex than this. The world we are attempting to model may include animated graphical objects, sounds, and composite objects, just to name a few. There is currently no relational database system that provides support for complex data types such as these. However, it should be noted that some relational database systems do support free form attributes, often called memos. These memo attributes are basically free form text fields of arbitrary length, so they could be used as a pseudo-repeating field. Also, as previously mentioned, some relational systems provide support for picture objects, but they are limited in their functionality.

b. Loss of Abstraction

To correctly model data to be stored in a relational database, there must be some form of decomposition of real world objects into flat objects. Occasionally, certain contrived relations must be created to achieve a satisfactory level of normalization. An example of this is a company database that contains employees, dependents, and possibly several other entities. Since **Dependents** is a weak entity related to an **Employee** entity, a possible method for implementing this relationship is to create a separate relation for each entity. In other words, create one relation to store instances of **Employees**, and another for instances of **Dependents**. This is an awkward way to model this real world relationship because Dependents have little

to do with the company database. A more appropriate way to model this relation and store the data would be with a multi-valued attribute of **Employee**; however, multi-valued attributes are typically not supported by RDBMSs. The attribute of **Employee**, call it **Dependents**, would store multiple instances of dependent entities. By storing the entities as an attribute, there would be immediate access to the dependant information whenever an employee object is retrieved.

To model this relationship in most RDBMSs requires two separate relations, an employee relation to hold instances of employees, and a dependent relation to hold instances of dependents. This greatly reduces the level of abstraction, and in some complex systems adds to the level of confusion. It also requires fairly complex algebra to retrieve some information. For example, if a user wishes to retrieve a list of Dependents for each female employee, five relational algebra statements are required. In the previous approach, only two are required, a selection of all female employees and a projection of the Dependents attribute.

c. Tuples Lack Function

It is attribute values that are stored in a relational database. While this is good for many bookkeeping applications, it is not desirable for many graphics based applications. In the average relational database, displaying a tuple or specific attributes of the tuple is a simple task that can easily be done by the database system. This is because the values are simple ASCII text or numeric values. Displaying a graphical object is not as simple. Consider the example of a multi-media database that stores animated pictures and sounds as well as textual or numeric data. If the user wants to display the animation sequence, the object retrieved had better have a function called **display** to play the animation because a non-application specific DBMS will not have the ability to do this. Also, if the user wants to play the sound clip that is stored in the object, then there had better be a **play-sound** function

because the DBMS does not have this capability either. A **display** method could also be written to display different things based upon the user's view of the object. This is not possible in a relational database because the information in the database does not carry any of the functions associated with the data. Therefore, current systems are only as good as the applications programs written for each specific application, as the data does not contain any information about how to handle itself.

d. Lack of Inheritance

Relational databases do not provide any means for defining attributes based upon previously defined attributes. Instead of being able to inherit previously defined attributes, the user must create new tables and new relationships between the tables to accomplish the same goal. For example, an employee database consists of employees who are all people. People all have the same general set of characteristics such as color hair, color eyes, name, age, etc. All of the employees, in addition to being people, are also members of a specialized group, such as hourly_employees or salaried_employees, with their own specific attributes. Using an object-oriented approach, each type of employee could be a descendent of the class People, and as such would have the characteristics previously mentioned. In a relational database, however, employee objects are represented as tuples in three different tables, one table contains all instances of people, another contains all instances of hourly_employees, and a third contains all salaried_employees. In an object-oriented approach there would only need to be two tables, one for hourly_employees and another for salaried_employees, because each type of employee would have all the attributes of people as well as the attributes specific to their class (i.e., hourly or salaried employees)

2. Object-Oriented Databases

Although, OODBMSs are a step in the right direction for database technology, they lack certain features which make them inadequate for tomorrow's applications. Specifically, the solid mathematical foundation and the standardization of the relational data model makes it a more attractive solution for many applications despite its limited modeling capabilities. If these deficiencies are overcome, there should be no reason why OODBMSs cannot gain the "lion's share" of the database industry, especially if they include the capability to efficiently model relational databases.

a. Lack of Mathematical Foundation

Relational algebra operations can be proven to be complete and correct based upon mathematical set theory, whereas OODBMS operations cannot. This is because there is no core of operations that has been thoroughly tested and proven for OODBMSs. This is definitely a limiting factor in OODBMSs because users want to know that they are definitely going to get the correct answer when the database is queried.

As previously mentioned, all relational operations can be built from the following operations: union, difference, projection, selection, and Cartesian product. These operations are based upon the ones developed by mathematicians, and can be combined to form all other types of relational operations. This provides the database user with a certain level of security as all operations conducted on a database are guaranteed to be correct and complete.

b. Lack of Standardization

The other major deficiency in current object-oriented databases is the lack of standardization of operations and class libraries, which is one of the strongest features of relational databases. Although each vendor's version of a relational

database system may be different, the basic operations are all there and the systems are therefore functionally equivalent. Such a core of operations does not exist in object-oriented databases because object-oriented programming still lacks a formal definition. Consequently some vendors provide languages that they call object-oriented simply because they contain support for objects or abstract data types, but not inheritance[1]. Once an industry-wide standard is adopted, a core of features can be required for OOPLs as well as OODBMSs and standardization will not be an issue.

All the OODBMSs surveyed for this thesis provide a class library, but each library is different. There is no standardization between systems, and one cannot count on all the classes used in one system to be available in the next. If there were a standard minimum set of operations required for all OODBMSs, and if it could be proven to be complete, then OODBMSs would almost certainly become more widely accepted.

c. Lack of Support for Relational Operations

This is understandable because object-oriented databases are not normally organized into logical relations. Object-oriented systems provide their users with a large storage area in which they can store any type of object. In the interest of compatibility and completeness, however, why not provide support for relational operations as part of the system? As this thesis points out, relational operations can be added to an object-oriented database, and in fact should be. Opponents of this idea might state that hierarchical and network databases are not currently supported, so why support relational? The answer to this is easy, they should all be supported [FN92]. The hierarchical data model lends itself to be easily

---

1. Systems supporting objects but not inheritance are more appropriately considered "object-based" [Wegn87].

integrated into an object-oriented database, and to a lesser extent so does the network. The flexibility of object-oriented languages and OODBMSs make it fairly easy to achieve this goal, but that is another issue. Compatibility with the relational model would definitely increase the credibility of OODBMSs. It would also open the door for developers to build relational systems for users in an OODBMS to allow for the future expansion of the database application as the needs of the customers change.

3. Relational/Object-Oriented Management System

The Relational Object-oriented Management System (ROOMS) as proposed in [Nels88, NMO90] is the model that this thesis is built upon. This system, however, lacks secondary storage features and this short-coming is the reason this thesis was written. The implementation proposed in this thesis adds the capability to store data to secondary storage, turning the ROOMS concept into a true database system.

4. Desired Properties of Database Systems

An extended relational system that could provide the ability to define complex data types such as sounds, pictures (including animation), and composite objects would be a good start. However, relational databases will never be a complete solution as they lack the ability to store functionality associated with the objects that they store.

Object-oriented databases need only add support in their class libraries for relations, records, and relational operations. The five basic relational operations should be supported in every OODBMS. This adds to the credibility of the system, and would help to standardize these types of databases.

B. WHY A RELATIONAL/OBJECT-ORIENTED DATABASE

The main reason that a relational/object-oriented database is important is because the world consists of objects that can and should be represented in a state that more closely represents their true structure. An object-oriented database can represent objects; however, current object-oriented database systems lack the mathematical foundation relational databases provide. Relational databases lack the modeling power of object-oriented databases, however, they provide a rich set of operations and rules for manipulating data. A combination of the two is a logical step in the advancement of database technology.

An example of how a relational/object-oriented database could be used was presented in [Nels88] and bears repeating. The example is a real estate database. It was pointed out that most real estate databases are already maintained by relational database systems. However, if a realtor wants to maintain all of the textual data regarding a piece of property, as well as blueprints and drawings of the house, most relational databases fall short. Also the database is not capable of storing maps of the surrounding neighborhood so that prospective clients can see what the general layout of the neighborhood is like, where the schools and churches are, and so on.

In a relational/object-oriented system, each record for a piece of property could have a blueprint stored in it as well as coordinates of the property that could be cross referenced to an area map. Depending on the query, a textual listing of the property information could be generated, or a picture could be displayed, or a combination of the two could be displayed. Also if the realtor wants to give the prospective buyer a map showing the property and its surrounding neighborhood, a map could be printed by performing a query on the table holding the property listings as well as the objects containing maps of the city.

C. WHY THIS WAS THE APPROACH TAKEN

This thesis extends the ROOMS concept into a full-fledged R/OODBMS. The R/OODBMS is implemented in Prograph, an OOPL that includes built-in database primitives for storing and retrieving objects in secondary storage. An R/OODBMS is also currently being implemented in a commercially available OODBMS [Spea92]. That implementation (when complete), along with this implementation, will complete the proof of concept of the feasibility/viability of an R/OODBMS which can meet the needs of both relational and object-oriented users.

Prograph was chosen as the language for this implementation primarily because it does offer built-in database primitives, as well as other desirable features. This is favorable because all of the file creation and accessing is done through these primitives, allowing the design and implementation to focus on the relational/object-oriented database concept rather than being bogged down with the details of reading and writing information to the disk. Some features were exploited and are discussed in the following chapter, however, these features can easily be replaced if desired. Another reason for choosing Prograph is because it exists on the Apple Macintosh which has a standardized operating system and routines available in ROM to handle the manipulation of pictures and sounds much more easily than any other platform.

IV. AN R/OODBMS IMPLEMENTED IN PROGRAPH

This chapter presents the design decisions and basic assumptions made in the development of this relational/object-oriented database system. The structure of the database is described and the Prograph classes designed for this thesis are explained in detail. Source code for all classes is contained in Appendix C.

A. BASIC ASSUMPTIONS

This database is built around six basic assumptions:

      1). the database applications will be developed in Prograph

      2). a persistent called current DB exists[1]

      3). the records stored in each relation must be instances of the same type

      4). all records being stored in a relation must be descendents of class **Record**

      5). every relation contains at least one key

      6). every relational operation returns an instance of **Temp Relation**

Many design decisions affecting this implementation rely heavily on these assumptions holding true. The following sections give more detailed descriptions of why these assumptions were made and how they affect the database design.

1. Applications Will Be Developed In Prograph

This assumption is made because the only way to design classes in Prograph is through the Prograph environment. Alternatively, it is possible to design a **Record** class that contains an attribute which is a list; this list could be then be treated as a "set" so that all the attributes of the record could be stored in a particular position in the list. This would allow users to build relations without going into the Prograph environment, as long as an appropriate interface is provided. However, it

---

1. This persistent can easily be created when the R/OODBMS application is created.

would also reduce the types of objects that could be stored in a relation because users would be unable to develop their own classes to be stored as records. Using the list approach also reduces the objects being stored to values only. Therefore, a major benefit of storing objects is lost because the value-only objects lack function.

Access to the Prograph environment is also required because some methods of **Record** and **Relation** should be over-shadowed for user-defined objects to be handled properly. The class **Record**, for example, makes some assumptions that are only true for simple objects. An example is the *keys* method of **Record** which defines every attribute of the user-defined record to be a key. This is a safe assumption if all the attributes are simple such as a strings or real numbers. However, if the attribute is not a simple type, it cannot be a key in a Prograph database. For example, an attribute that is an instance of another class could not be used as a key value because keys in Prograph are limited to boolean, integer, natural, real, and string.

2. Current DB Persistent

A persistent is assumed to exist called **current DB**. It is used to allow locals within methods to open the database tables without requiring the method to have an input of the current **Database** instance just to pass it to the local. Since access to the database file is achieved through the database id, and access to the Structure Table is through the Structure Table id, both pieces of data are required to open an existing relation. It was decided that the logical course of action is to use the persistent rather than clutter every method with extra terminals.

This assumption is related to the database application and as such it is assumed that the database developer will provide this persistent, or selectively load it. The second reason for creating a current DB persistent is because Prograph allows

multiple database files to be open, and current DB provides a convenient place to store an instance of the database currently being manipulated.

3. Relations Contain Records of the Same Class

To model a relational database, all the records in a relation must be of the same class. The reason is because all records in a relation are made up of attributes, each with specific domains. Each tuple in the relation has corresponding attributes with the same domains. In general, it does not make sense to put different types of records in one relation; rather, they should be stored in different relations. For example, if the user decides to create two types of employee objects, say an *hourly_employee* and a *salaried_employee*, to be stored in a database, then there should be a class definition for each. If each type of employee is of a different class because they differ in some way, then they should not be stored together in the same relation. Even if they have exactly the same attribute types but function differently, they should be separated into two different relations. This is primarily a database application design issue, but this constraint is imposed to more closely adhere to the relational model.

4. User-defined Records Must Be Descendents of Class Record

This is a very important assumption, because when the end-user tries to create a new relation to be stored in the database the system looks for descendants of class **Record** and only allows the user to select one of these dependents to be stored in the relation. The system also names the relation based upon the name of the class being stored in the relation. In other words, if the user wishes to create a relation called **Person**, then there must be a class **Person** defined as a descendant of Record and there must not already be a relation called **Person** declared in the database. The assumption that relations are made up of records is used both directly and indirectly in many methods.

5. Every Relation Contains At Least One Key

As previously discussed in Chapter II, the only way to access a cluster directly is by using the cluster id, and indirect access is achieved through keys associated with clusters. In order to access the clusters in the database, a key called **~primary key** has been defined for all relations. The ~ is used to ensure that it will be the last key associated with a table; also, if the user wishes to define a key called **primary key** there will not be a name conflict. The value of this key is determined by the application programmer. The intended use for this key is to store the primary key (in other words a unique value associated with the record) of each record since every record must be unique in a relation. However, this key is not limited by this convention.

One use of the **~primary key** is to store a default value of 'a' or '1' for every tuple. This has the effect of ordering the keys in the order in which the records are written to the database.

6. Every Relational Operation Returns A Temp Relation

Every relational operation has to return the same kind of result for consistency. It does not make sense to have one operation return a **Relation** while another returns a list, or a **Temp Relation**, or something else. This is important when considering the Cartesian product and projection operations. These two operations do not normally return relations that look like any other relation.

The reason a **Temp Relation** is returned rather than a **Relation** is because tuples of a relation are actually stored as instances of the class that defines them, whereas the results of relational operations may not be instances of a specific class. **Temp Relations** are written to secondary storage as lists, because Prograph does not allow declaration of classes on the fly and because a Prograph list can have instances of classes or any other object as an element. For example, if the user performs a

Cartesian product operation on two relations, the result is a new relation with tuples containing all the attributes of the first relation and all the attributes of the second. There is no way to anticipate all the Cartesian product queries the user will perform, so there is no way to define a class for each Cartesian product result. Similarly, there is no way to anticipate all the projections a user will make, so classes cannot be defined for this operation either.

B. DESIGN DECISIONS

Before designing the classes for this implementation, some decisions as to the structure of the database had to be made. These included how to represent the relations and how to implement the relational operations.

It was determined that a class called **Relation** should be created. Instances of this class require only three attributes. First is the *relation name*, which is taken directly from the class definition of the class to be stored in the relation. Next is *attribute-names* which is a list of all the attributes of the tuples in the relation. The last attribute is *attr-types*, which is also a list containing the type of each attribute.

It was also decided that each relation should be stored as a Prograph database table containing instances of **Record**s as clusters, and that each instance of a **Relation** would be stored in a **Structure table**. The Structure table contains a single entry for each relation stored in the current database. These entries are instances of Relation and have a key value associated with them for fast retrieval. The key value is the name of the relation so all relations must have different names.

The decision was made to represent the relational operations as methods of the class **Relation**. In examples provided by TGSSystems with Prograph, they have implemented some of the relational operations as separate classes rather than methods of a class **Relation**. In examining relational databases and the relational model, it became apparent that the relational operations should be methods, rather

54

than separate classes. The relational operations are functions or operations that act upon instances of **Relations**, and as such should be methods of the **Relation** class rather than separate entities that act upon objects of type **Relation**. The latter approach is more of a structured programming approach rather than an object-oriented approach.

C.   THE STRUCTURE OF THE DATABASE

The database files created by applications using these classes all contain a Structure Table. This **Structure Table**, as previously mentioned, holds instances of every relation created for the database. This table is required because once a relation is created, information about it needs to be stored for future sessions. In a Prograph database, this information about the relation must be explicitly written to the database disk file or it will be lost from one session to another. This is because there is not a single large repository for all the database applications to use as in most OODBMSs. Each Prograph database is made up of two files, the database file and a key file, which do not have access to information about any other databases. Although Prograph database files automatically maintain the names of all the tables associated with the database, other information about the attributes in each relation is not maintained. Thus, without the structure table the user would have to create a new instance of a relation and compute the associated attribute characteristics every time a database is opened. Using the structure table approach, the user simply retrieves the relation instance from the table when it is required.

When a new relation is created, the relation bears the same name as the class whose instances will be stored as tuples in the relation. This class representing the tuples must be a descendant of the class **Record** for this schema to work. Once this is done a Prograph table named for this class can be created and tuples can be written to the disk.

Every **Relation**, **Temp Relation**, and tuple is written to the disk file rather than being stored in memory. This feature provides consistency and some level of security for the data being stored. Previous implementations of ROOMS did not have secondary storage features and therefore the data lacked any permanence. This implementation has features that allow the use and manipulation of the data that is stored to a disk file.

D. REQUIRED CLASSES

To implement a relational/object-oriented database in Prograph four classes were designed. First is the class **Database**. Instances of this class are created for every database that is opened by the user. This is necessary because Prograph allows multiple databases to be opened at the same time, as well as a single database being opened by more than one application. The **Database** instances maintain the path to the open data file, as well as the information about the database file (i.e., the file name and the volume id as well as the Structure Table id).

The next class that was required is the **Relation** class. This class is the workhorse of this design, containing attributes describing the keys to access the tuples of the relation as well as the path to the database. Most importantly, it also contains all of the methods required to perform relational operations on the data in the database. A descendant class of **Relation** is **Temp Relation**. This class represents derived relations and modifies some of **Relation**'s methods to handle these temporary relations.

The last class required by this design is the class **Record**. This class is the root of the sub-tree containing user-defined classes that will represent the tuples of a relation. Each of the afore-mentioned classes are now described in greater detail.

1. Database Class

This class contains basic methods required to create and access a Prograph database. A database object contains the following instance attributes: *file name*, *file volume id*, *database id*, *Structure Table id*, and *temp relation list*. *File name* and *file volume id* exist primarily to handle the opening and closing of the physical disk file (necessary for multiple access to the file). The attribute *database id* is a pointer (or path reference id) to the database file. *Structure Table id* holds a pointer to the table in the database that contains all the relations associated with the database. Finally, *temp relation list* is a list of temporary relation names created by user queries. The temporary relations (which are simply Prograph tables) are cleaned up by the *close-db* method.

Some of the methods required for this class are *new-db*, *open-db*, *close-db*, *new-relation*, and *display-yourself*. There are several other support methods not critical to the design of the overall system. The first method, *new-db*, presents the user with a standard Macintosh file creation dialog box and then creates the database files (database and key file) using Prograph database primitives. It also creates an instance of **Database** and a table in the database file to maintain the structure of the database (the Structure Table). Once the **Database** instance has been created, it is stored in a persistent named **current DB** to eliminate the need to constantly pass database instances from one method to another. There is also some error checking included to ensure that no NULL files are created.

The next method in this class is the *open-db* method, which is responsible for opening a database file. It has no inputs because it uses Prograph primitives to display a standard Macintosh file selection dialog box which allows the user to find and select the database file to open. It then prompts the user to select the mode in which to open the database (either share, query, or update mode). Once the files are

opened, an instance of **Database** is created and placed into the **current DB** persistent and the database id is passed as the output of the method. Again there is error detection and handling to prevent the user from opening a NULL file. If a NULL file is selected this method generates a failure control and passes it to the calling method to be handled.

Next is the *close-db* method, which removes the selected **Database** instance from the **current DB** persistent, deletes any temporary relations, then closes the database file. This method also requires no input and produces no output as there is currently only support for one active database at a time.

A *delete-db* method is also provided to remove a database from the disk. This method presents a standard file selection dialog box and once the selection has been made, the user is asked to verify the delete. If the user acknowledges the delete, then the database file and the key file are permanently deleted from the disk.

2. Relation Class

There are three instance attributes of this class; *relation name, attribute-names,* and *attr-types. Relation name* contains the name of the class stored in the relation. *Attribute-names* is a list of attribute names of the class being stored in the relation. The value of this attribute is taken directly from the user defined class, which makes the application programmer responsible for putting in the correct attribute names. Since each tuple is made up of one or more attributes, this list is used to determine "union compatibility" and to verify selected attribute names for projections and selections actually exist in the relation being queried. *Attr-types* serves the same type of purpose as *attribute-names*, except that the elements of its list are the attribute types (i.e., string, integer, etc.) of the entries in *attribute-names.*

This class contains many methods, some of which are required by other methods and some which exist to reduce redundant code in other methods. The

major methods required are: *select-relation*, *add-tuple*, *remove-tuple*, *display relation*, *union*, *projection*, *selection*, *difference*, and *Cartesian product*.

The method *select-relation* takes as an input a **Database** instance and presents the user with a dialog box asking which relation to open. Once the user makes a selection, the corresponding **Relation** instance is retrieved from the structure table and returned to the caller. The purpose of this method is to retrieve **Relation** instances from the database file for use in the application.

The *add-tuple* method takes as input an instance of a **Relation** and the tuple (object) to be added. The tuple is checked to ensure it is of the correct type to be stored in the relation. Then all of the key values are extracted so the appropriate key-cluster associations can be made, including the ~**primary key** value. It makes use of a method defined for all **Records** called *get primary key* to get the value of the primary key.

The *remove-tuple* method takes as inputs the **Relation** instance and the tuple id, and has no outputs. It is assumed that the user will have already selected the tuple to be deleted prior to calling this method.

Display-yourself is a method that displays the relation one record at a time. This is a method that can be overshadowed to suit the needs of the application. There is one input, the instance of the relation, and no output because the output is directed to the screen.

The next five methods to be discussed are the relational operations. These methods are able to handle any type of object that can be defined in Prograph. The application programmer can, however, re-write methods for any of the basic five operations to take advantage of the keys declared for their objects. For example, the select operation may be too slow because it looks at each record and compares the attributes to see if they are equal. If the attributes in question were stored as keys, search times would most likely decrease.

The three inputs to every relational operations are: a **Relation**, another **Relation** (or **Temp Relation**) or a list, and the resulting relation's name. Every relational operation also outputs a **Temp Relation**, which is stored as a table in the Prograph database file with the resulting relation's name as the name of the table.

The *union* method opens both input relations and then writes every tuple of the first relation to the **Temp Relation** created. It then reads each tuple from the second relation and searches the first relation to see if their is a match. If a match is found, then it is not included in the **Temp Relation**, otherwise, it is added to the **Temp Relation**. This is done to ensure the uniqueness of every tuple in the result relation.

The second input to the *projection* method is a list of attributes. The attributes in this list are compared to *attribute-names* in the input **Relation** instance. If they are valid, the method proceeds to read every tuple in the relation, writing only the values of the attributes requested to the **Temp Relation**.

The second input to the *selection* method is the selection criteria in the form of a list containing the attribute for the comparison, the operation (=, >, etc.) and the value. The method calls upon the user-defined methods for equality (equal? greater-than?, etc.) to determine which tuples should be put into the **Temp Relation**. Default equality methods are defined in the **Record** class, but they are fairly simple as it is impossible to anticipate every object defined for a record.

The *Cartesian product* method reads in a tuple from the first relation, converts it to a list, and then converts every tuple in the second relation to a list and appends them to first list. In other words, if the first relation has two tuples a and b and the second relation has three tuples x, y, and z, then the result of the Cartesian product will be a relation with six tuples: (a, x), (a, y), (a, z), (b, x), (b, y), (b, z). This method uses the ~**primary key** to step through each relation. This implementation

could be looked at as two nested "For-statements" in a structured programming language.

The *difference* method opens both relations, examines each tuple in the first relation and compares it to every tuple in the second relation to see if they match. If they do match, the tuple is not included in the resulting **Temp Relation**, otherwise, it is included. The comparison in this method is based upon the user-defined (or default) equals? method in the user-defined record class.

3.   Temp Relation class

This class defines no new attributes; however, all relational operations are redefined. The reason these methods are redefined is because the **Relation** versions of these methods make the assumption that the attribute names are the names of actual attributes of a class. In contrast, the attribute names associated with **Temp Relation** objects are merely ways to locate the attributes based upon their position in the list that they have been stored in. For example, if a **Temp Relation** is created and has an *attribute-names* value of (lname, age), then a tuple stored in the Temp Relation could be (Filippi, 27). To retrieve the age attribute of this particular tuple the position of the attribute named "age" would have to be determined from the *attribute-names* list. Since the value of the position of the "age" attribute is 2, a get-nth (with n=2) could be performed on the tuple in question and the value 27 would be returned. In the **Relation** version, to retrieve the attribute age from a person record a "get" method is performed on an instance of person class with the attribute name "age" injected into the operation.

All the changes to the relational operations have to do with how attributes are retrieved and how they are accessed. The algorithms for performing the operations are the same, so they will not be discussed further.

4. Record class

The **Record** class has three class attributes: *attr-types*, *attr-names*, and *keys*. These attributes can either be set directly by the programmer when the classes are defined (i.e., default values can be set), or they can be defined when they are accessed. The purpose of the first two attributes is to provide the **Relation** class attributes the values they need to keep track of the attributes of the tuples being stored. The *keys* attribute is used to create keys for the relations in addition to the ~**primary key**. For example, the programmer may design records with an attribute X that is the primary key, but know that many queries will be performed based upon a non-unique attribute Y. Then the attribute Y can be declared as a key by placing the attribute name Y in the *keys* attribute. Then when a *selection* method is written to overshadow the *selection* method of **Relation**, the key Y can be used to retrieve values from the relation, rather than reading every tuple to determine whether the condition is met. This is useful if the database application programmer wants to create a descendant of **Relation** and overshadow some of the relational operation methods. The applications programmer can then write these methods in such a way as to take advantage of the key and speed up the query.

The methods provided with the **Record** class are *attr-types*, *attr-names, keys, equal?, display-yourself,* and *get primary key*. The first three methods are set methods[2]. These methods can be overshadowed by the programmer if deemed necessary. The *keys* method treats every attribute as a key and sets the value of the *keys* attribute to a list containing the attribute name of all the attributes of the class. This is fine if all the attributes are simple, but if the attributes represent a list or an instance of another class, they will not make valid keys and the system will most likely crash.

---

2. Set as in "set" or "get", vice "set theory".

The *equal?* method evaluates two records and determines equivalence. If the objects are composed of simple attributes, this implementation of *equal?* works fine because the objects entering the method are converted to lists, and the lists are compared with the "=" primitive. For more complex objects an *equal?* method should be defined by the application programmer.

Display-yourself is a method that simply invokes the display primitive. If the record contains special types of objects such as pictures or sounds, it may be more desirable to create a *display-yourself* method with the appropriate interface to best display the object. Even for simple objects, a display window could be designed to make the result more aesthetically pleasing.

The *get primary key* method must be over-shadowed unless the first attribute defined for the user-defined class is to be the primary key value. This method is designed to get the value of the first attribute declared in the class and use it as the primary key. If the primary key value is to be something more than a single attribute, the method should be over-shadowed. An example is, a company database that contains a relation between employees and projects. The combination employee number and project number might be the only way to distinguish one tuple from another (i.e., the primary key). If this is the case the *get primary key* method could be written to get the values of employee number and project number, concatenate them together and return the catenated value as the output of the method.

V. SUMMARY, CONCLUSIONS, & SUGGESTIONS FOR FUTURE RESEARCH

The purpose of this research was to design and implement a relational/object-oriented database system with secondary storage features. This was a continuation of a previous research project and has satisfactorily demonstrated that the concept of a relational/object-oriented database management system as proposed in [Nels88, NMO90] can be extended to a system with secondary storage.

A. SUMMARY

A detailed literature review was accomplished in which object-oriented programming, databases, and Prograph were investigated. The feasibility of creating a relational/object-oriented database system was determined, and an implementation was proposed and implemented.

The implementation is language dependant, but shows that the concepts are valid for any OOPL with secondary storage features. Therefore, the concepts proposed are also valid for an OODBMS (which can be considered to be specializations of OOPLs).

B. CONCLUSIONS

OODBMSs can be used to store and manipulate relational databases. Since it was shown to work in a lisp-based language designed to be bolted on top of an OODBMS [Nels88, NMO90], and it has now been shown to work in an OOPL with secondary storage features, it is safe to say that it should work within any OODBMS. Since OODBMSs contain all the features of OOPLs, along with the ability to handle the storage and retrieval of data to and from secondary storage devices, they should therefore be able to handle both relational, and object-oriented data within the same system.

C. SUGGESTIONS FOR FUTURE RESEARCH

Future research in this area should include, but is not limited to, the following areas: implementation of relational operations in a commercially available OODBMS, optimization of relational operations in an OODBMS, allowing complex objects to be keys, declaration of a standard class library for OODBMSs to include the relational operations, and extension of the ideas presented here to include addition of other types of databases to OODBMSs.

1. Implement Relational Operations in a OODBMS

Although this project implements a relational/object-oriented database, the final proof of concept is whether or not this idea can be implemented in an existing OODBMS. The class libraries available in OODBMSs are usually very rich and should provide a means for implementing the ideas posed herein in a much more elegant and efficient way.

Once an OODBMS implementation is completed, the proof of concept for adding relational operations to an object-oriented database management system will be complete.

2. Optimization of Relational Operations

The relational operations presented in this thesis were written to prove that relational operations could be added to an object-oriented system with secondary storage features. Although this was demonstrated, the time required for some queries on large relations is unacceptably slow. This is because the relations are retrieved from secondary storage and then written back to secondary storage in a different form for every query. Disk access time, even on a fast system, makes these operations perform at a less than desirable speed.

3. Allow Complex Objects to be Keys

In this implementation the values of keys are limited by the language to booleans, integers, real numbers, or strings. In a general purpose R/OODBMS it is important to allow arbitrarily complex objects to be key values in a relation. Limiting keys to simple data types limits the usefulness of the system in much the same way as the limited number of data types available in conventional relational database systems limit their use.

4. Addition of Other Types Of Databases To OODBMSs

Although relational databases can be represented in an OODBMS, what about hierarchical or network databases? Initial work has been done in this area [FN92b], but a more detailed design and implementation should be attempted to determine whether OODBMSs are able to efficiently handle the various data models. The implementation of a hierarchical database and a network database within an OODBMS should sufficiently prove the point that OODBMSs are indeed sufficient to allow the storage and retrieval of all types of database systems.

5. Standardized Class Library For All OODBMSs

Once a determination as to the feasibility of implementing all the current data models within the confines of OODBMSs is established, a logical step would be the standardization of the class libraries associated with each vendor's OODBMS. Just as SQL is the standard for relational database languages, a standard minimum requirement should be developed for all OODBMS vendors. This would ensure that regardless of which OODBMS you had access to, you would still be able to use it to store and manipulate the type of data in any way desired.

APPENDIX A - CREATING A SAMPLE DATABASE APPLICATION

A. BASIC ASSUMPTIONS

It is assumed that the user has a working knowledge of Prograph and database design issues. The primary purpose of this appendix is to show how to utilize the tools provided in the classes written for this thesis.

B. DESCRIPTION OF THE METHODS PROVIDED

Appendix B presents all of the methods in each class in their graphical representation. The inputs and outputs for the methods that will be directly called by the user are annotated in the graphical representations. All of the methods that should be called by the user are briefly described in the following paragraphs. For a more detailed description of each method see Chapter IV and Appendix C.

1. Database Class

The following methods of the Database class should be used by the applications programmer to create a database application.

a. new-db

Used to create new database files (the database and the keys file). This method fails when the *db-new* primitive fails.

b. open-db

Used to open an existing database. This method fails if the *db-open* primitive fails.

c. close-db

Closes an open database.

d. *delete-db*

Deletes an existing database from secondary storage.

e. *rename-db*

Rename a database.

f. *display-yourself*

Uses the show primitive to display a list containing the names of **all** relations in the Database.
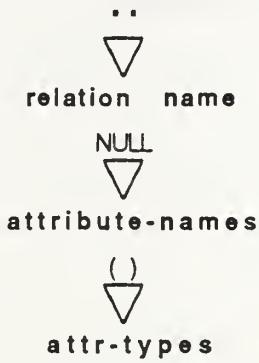
g. *new-relation*

Creates a new relation based upon the sub-classes of **Record**. This method fails if the database is not opened in update mode.

h. *delete-relation*

Provides the user with a dialog box from which a relation can be selected for deletion.

2. Relation Class

The following methods should be used by the applications programmer to create and manipulate relations in a database.

a. *select-relation*

Presents a dialog box containing every relation that exists in the current database, returning the relation selected.

b. *add-tuple*

Adds a tuple to a relation. The inputs are an instance of **Relation** and an instance of a user-defined record (tuple).

c. remove-tuple

Deletes a tuple from the relation. It is assumed that the cluster id for the object has been retrieved so the cluster (tuple) can be deleted.

d. union

Performs a union of two relations. Every tuple from the first relation is written to the **Temp Relation**, and every tuple from the second is evaluated to see if it is the same as one from the first relation; if it is, it is not written to **Temp Relation**. This method fails if the two relations are not union compatible.

e. selection

Retrieves every tuple from the relation whose attribute(s) satisfy a selection condition and places the results in a **Temp Relation**. If the attribute(s) used in the selection condition does not exist in the relation, this method will fail.

f. projection

Retrieves only the requested attribute(s) from every tuple in the relation, placing them in a **Temp Relation**.

g. Cartesian product

Produces a **Temp Relation** that contains tuples created by appending the value of every tuple from relation number two to each tuple of relation number one. The **Temp Relation** has as many attributes as the sum of the two input relations and as many tuples as the cross product of the relations.

h. difference

Produces a **Temp Relation** with all of the attributes that are in the first relation not in the second relation. This method fails if the two relations are not union compatible.

i. *display-yourself*

Uses the system defined primitive display yourself to display every tuple in the relation.

3. Temp Relation Class

The methods of this class have the same functionality as the **Relation** class methods of the same name. The only difference is that these methods have as the first input to each relational operation an instance of a **Temp Relation**, whereas the **Relation** counter-parts have a **Relation** as the first input to the relational operations.

4. Record Class

These methods will most likely be over-shadowed by descendents of **Record**. However, since these methods must be present in some form, they are included in the class definition of **Record**.

a. $=, \neq, <, >, \leq, \geq$

All of these methods take two tuples and compare them. The comparison operations provided here are based upon the tuples containing simple attributes, and the attributes values are compared to determine equality.

b. *display-yourself*

This method uses the system defined *display* primitive to display the record.

c. *get primary key*

This should definitely be over-shadowed unless the primary key happens to be the first attribute declared in the record class.

C. BUILDING A DATABASE APPLICATION

1. A Simple Application

The sections that follow will take you through creating a **very** basic database application which can be run in the Prograph Editor or compiled into a stand alone application.

2. Basic Description of the Application

We will be creating a simple database of people records. There will be two kinds of people, students and teachers. Each type of person has the attributes: *last name, first name, age, sex, address,* and *widget.* The first four attributes are simple attributes, the *address* attribute is another class containing the attributes *number, street name, city, state, and zip code.* The *widget* attribute is an arbitrarily complex object.

Since we have two different types of people, we will create the class **people** with the descendants **students** and **teachers**. Since Prograph does not currently support multiple inheritance, **people** must be a descendant of **Record**, and **students** and **teachers** must be descendants of **people**.

We will now show how to create a single database to hold the relations **students** and **teachers,** and how to perform relational operations on these relations.

3. The First Step

The first thing to do after opening Prograph is open the Classes window by selecting it from the Windows menu. Next, selectively load all the classes from the file Thesis.pgs. This is done by selecting Open from the File menu. When the open file dialog box appears click on the Selective Load check box (this will also automatically select the Incremental Load check box), navigate to the file Thesis.pgs and choose the Open... button. This presents a list of all the classes in the file. Select

71

every class by holding down the shift key and clicking on every class name (all of the current Prograph System Classes are included in this file). Now do the same thing with the Universal methods. Open the Universal window by selecting Universal Methods from the Windows menu and repeat the steps above for opening the Thesis.pgs file. Instead of a list of classes in the selection window, a list of all the methods in the file is presented. Select only the methods named **make list, make class list**, and **read & write**. The final thing to do is open the Persistents window by selecting it from the Windows menu and create a new persistent by clicking in the white space of the window. When the new persistent appears, name it current DB. It is very important that it be named correctly, note that Prograph is case sensitive, so it must be typed exactly as it appears.

4. Create The Classes

Open the Classes window (which looks like Figure A.2). Click anywhere to create the **address** class. Once a class icon appears in the window, type **address** and then double-click on the left side of the icon to open the attributes window for the class. This is where the attributes *number*, *street name, city, state,* and *zip code* are created and given default values. Click in the white space of the attributes window to create a new attribute. When the icon appears, type in the attribute name (i.e., *number*). Do this for every attribute. After creating all of the attributes, close the window so that the Classes window is in view.

Repeat the above procedure for the **people** class. Except this time after creating and naming the *address* attribute, double click on it. A window opens that allows the value of the attribute to be set. Scroll up the list on the left to the word address. Select address by clicking on it and then press the **OK** button. The default value of the attribute *address* is now set to be an instance of the class **address**.
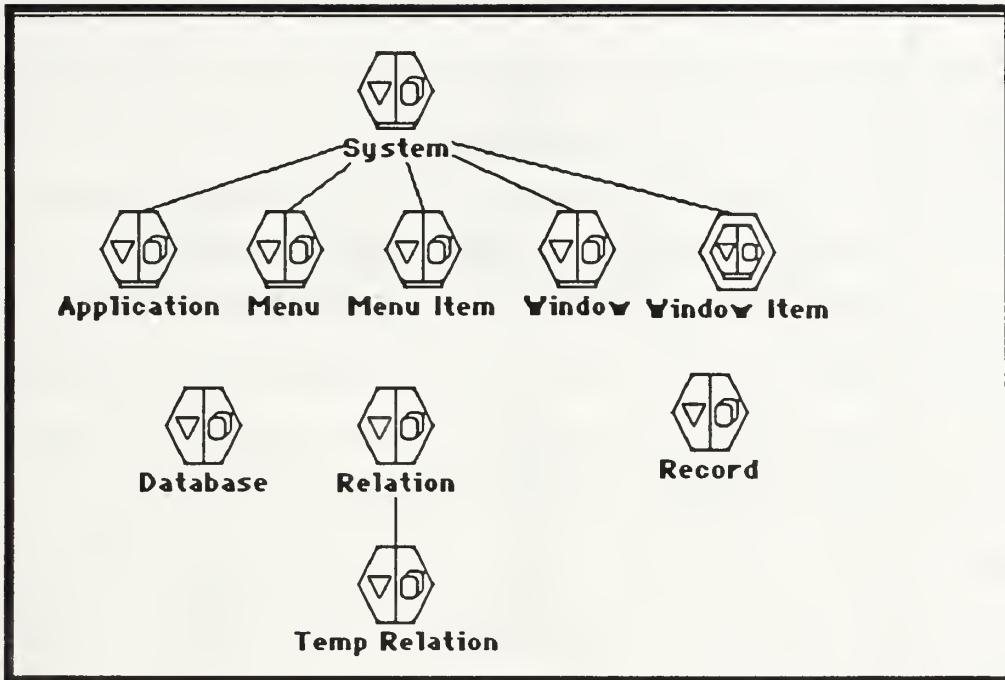
Figure A.1 Classes Window

Next, create a sub-class of people. Begin by clicking on the **people** class icon. Then while holding down the option key, move the mouse to some white space and click the mouse button. A new class icon will appear with a line connecting it to the class **people**. Type the word **students** to name the new class, then repeat the procedure and create a new class called **teachers**. Once this is done the Classes window should look similar to Figure A.2. Now that all attributes for students and teachers are finished, the only thing left to do is to attach the class **people** to the **Record** class icon. Do this by clicking on the **Record** class icon until it highlights, hold down the option key, and click on the people class

The user interface can now be designed as in any Prograph application. The next few paragraphs present a very basic interface; for more detailed information refer to [TGS88a, TGS88b, TGS91].
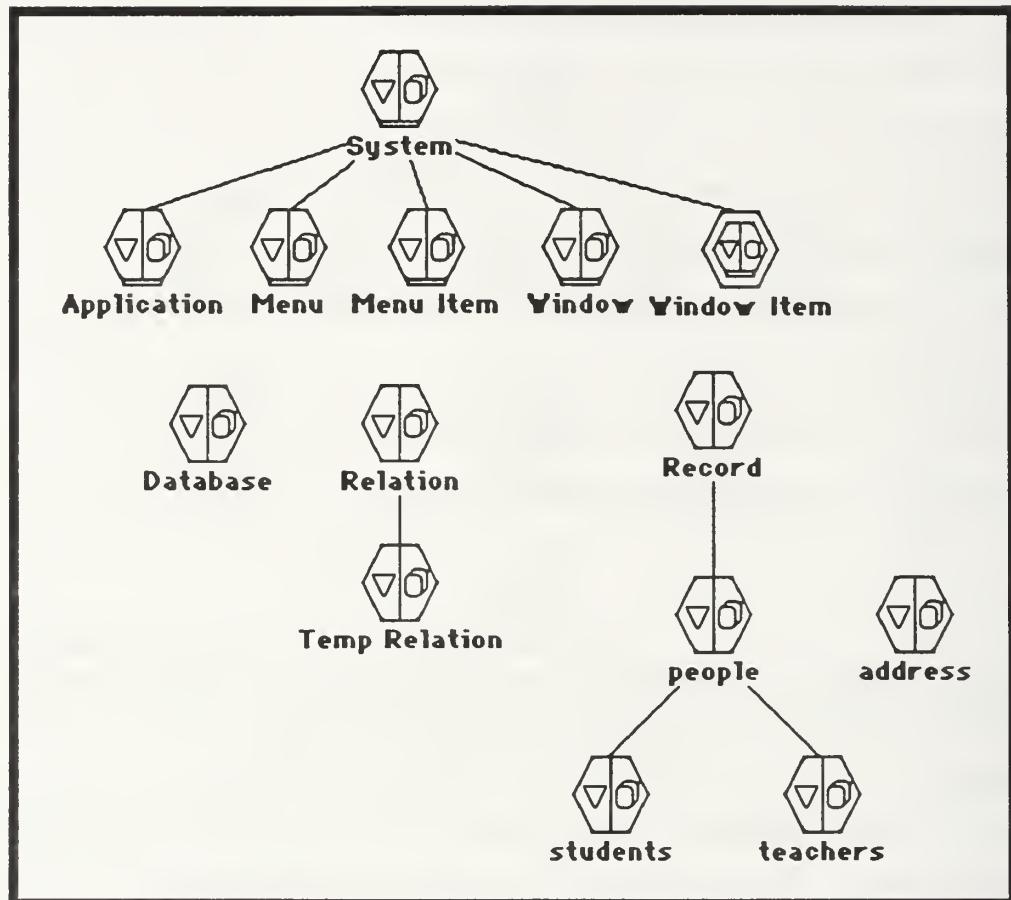
Figure A.2 After Building New Classes

5. Creating the Interface

Keep in mind that the windows to be used for data entry, the windows to be used to display the results of queries, and the query building window must be developed. Since this application has relatively few queries that will be performed, we will create a button for each specific query (e.g., a button for selection, a button for projection, etc.). Another possibility would be to provide an interface that allows the user to build queries dynamically.

74

a. Creating the Query Interface

To create the query interface, select **Edit Application** from the **Exec** menu. This brings up a window that allows windows and menus for the application to be created. We will now create a window that contains five buttons representing the five basic relational operations. Select **Window** from the **Classes** scrolling list on the lower left-side of the window, and then click on the **>New Instance>** button below the list. This creates a new instance of window named **Untitled.** Double-click on **Untitled** to open it for editing. Once the window is created, click in the white space to create a new object. Once the object appears, double-click it to open a dialog box to set the type of the object. When the dialog box opens, **Button** is selected by default so all that is required is to click the OK button. Once this is done, another dialog box opens which allows the button to be named and a method associated with it to be called when the button is clicked. Name the button **Selection.** There is also a field for **Click Method**, type in "Selection", and click the **OK** button. This button, when clicked, executes a universal method called **Selection**. Create the other four methods just like the Selection button. Once they are completed, hold down the option and command key and double click on the white space in the window. This brings up a dialog box that allows the window to be named and other parameters to be set. Type in "Queries" in the **Window Title** field then click **OK**. The interface for the queries window is now complete.

b. Creating the Data Entry Interface

To create an interface to allow the user to enter data into the database, create a new window as previously described. Rather than **Buttons**, create **Edit Text** objects on the window, one for each attribute of people objects (including the five for address) except for the complex object *widget,* and name the objects the same names as the attributes they represent. These Edit Text objects will be used by the

end-user to type in the values to be stored in the relation. Depending upon what type of object a *widget* is, an interface specifically designed for the widgets would have to be created. Also create a button named **Student** and another named **Teacher**. Once this has been completed, name the window (just like the previous window) "Data Entry" and then close the window.

Now select each window from the scrolling list on the **Application** window and press the **Add To Active List** button. This adds the new windows to the application, and these windows will be opened upon start-up of the application.

c. Creating the Menus

Finally, a menu should be added to the application that allows the end-user to open and close databases as well as select relations to open. To create a menu, select **Edit Application** from the **Exec** menu. When the window opens select the **Menus** radio button and create a new instance (just like when the window instances were created) and open it for editing. Name the menu "Database", and tab to the **Item** field. Type "New DB" in the **Item** field. Tab to the Method field and type "New DB", then click the **Insert After** button. Repeat these steps for "Open DB" and "Close DB". Click the **OK** button when finished. Now create a menu called "Relations", and create menu choices for "Open Relation" and "New Relation".

Now return to the **Application** dialog, select the menus just created, and click the **Add To Active List** button. This adds the new menus to the applications menu bar when it is run.

D. COMPLETING THE DESIGN

Now that the interface has been built, run the application and build the methods for each button and menu item created while the program is executing. To do this, select **Run** from the **Exec** menu. Both windows should appear, unless they were

designed on top of each other. If this is the case, select the title bar of the foreground window and move it to expose the other window.

Once both windows are visible, select **New DB** from the **Database** menu. A dialog box is presented stating that the universal method "New DB" does not exist, do you want to create it? Respond by clicking the **OK** button. This will open a method window with a dotted background. Double-click in the window to open its editable case window. Now create an operation to call the *new-db* method in the **Database** class. This method does not require any input terminals, and when Prograph menu items call a method they always send it the instance of **Menu**, the **Menu Item**, and an **Event Record**. Close the case window and press return to activate the operation that was just created in the method window. When this operation executes it will present the dialog box previously describe for the *new-db* method. Now do the same thing with the **Open DB** and **Close DB** menu items.

Repeat the above procedures for the **Relations** menu except that when adding the operation to each case window, it will be necessary to create an additional persistent operation to get the value of **current DB** to feed into the *Relation/select-relation* method (to open a relation) and *Database/new-relation* (to create a new relation).

Now that the menus are done, the same type of thing must be done for the buttons in the windows. The **Queries** window buttons will all require the user to specify which relation/relations will be operated on, and what the resulting **Temp Relation** will be called. The values can be obtained by using *select-relation* to get the relation instances, and the **ask** primitive can be used to get the result name as well as the list of attributes or selection condition depending on the relational operation being performed. The root of the relational operations can be attached to the *Temp Relation/display-yourself* operation (which will display every tuple in the

Temp Relation). In a more sophisticated implementation a window with scrolling fields or other features would be used

The **Data Entry** window has only the two buttons **Student** and **Teacher**, and the methods for these buttons can be created the same way as the previously discussed buttons and menu items. For this implementation the buttons will both make use of the *Relation/add-tuple* method to add the tuples to the database. They will differ in that each will create an instance of the appropriate class (either **students** or **teachers**) and this instance will be written to the correct relation by the *add-tuple* method.

Once all the buttons and menu items have been created and fully implemented a very simplistic application has been constructed. It could be compiled into a stand-alone application by the Prograph Compiler or it can be used in the interpreted mode. Naturally, some of the other features provided have not been discussed, but could be easily implemented.

E.   DETERMINING WHICH METHODS TO OVER-SHADOW

1.   Database Class

The only method that might require over-shadowing is the *display-yourself* method. This method displays a list containing every relation defined for the database. It uses the **show** primitive to display the list of relation.

2.   Relation Class

The *display-yourself* method could be over-shadowed by the programmer because the current implementation reads every record in the relation and displays them with the display primitive. Some of the relational operations could also be over-shadowed (as mentioned in Chapter V) to take advantage of keys associated with the user-defined records.

3. Temp Relation Class

None of the Temp Relation methods should be over-shadowed.

4. Record Class

Virtually all of the methods associated with **Record** can be over-shadowed. The purpose of this class is to act as a guide for developing new records, but the methods make no assumptions about the structure of user defined record. The only assumptions made in these methods are that the class attributes contain the appropriate values. What values go into these attributes are completely up to the designer.

The equality operations ($=, \neq, <, >, \leq$, and $\geq$) should be over-shadowed to correctly handle comparisons of the objects. If these are not over-shadowed, some relational operations may not return the expected results.

Display-yourself should be overshadowed to properly display the objects. If it is not over-shadowed, the **display** primitive is used. This method should be over-shadowed as part of the user interface design so when the records are displayed, your windows are used.

APPENDIX B - ATTRIBUTES AND METHODS

▽ Database

```
          ..
          ▽
      file  name
          0
          ▽
    file  volume  id
          0
          ▽
     database  id
          0
          ▽
  Structure  Table  id
          ( )
          ▽
  temp  relation  list
```

Database

new-db	Input: None Output: DBId
delete-db	Input: None Output: None
new-relation	Input: <<Database>> Output: <<Relation>>
open-db	Input: None Output: DBId
rename-db	Input: None Output: None
delete-relation	Input: None Output: None
close-db	Input: None Output: None
display-yourself	Input: None Output: None Uses the show primitive to display a list containing the names of all relations in the Database
temp relation list	

▽ Relation

..
▽
relation   name

NULL
▽
attribute-names

( )
▽
attr-types

🔳Relation

open table
Input: <<Relation>>
Output: TableId

open to first tuple
Input: <<Relation>>
Output: Primary key Id, Table Id

select-relation
Input: DBId
Output: TableId

display-yourself
Input: <<Relation>>
Output: None

make Temp Relation
Input: name, <<Relation>>
Output: <<Temp Relation>>

add-tuple
Input: Table Id, tuple
Output: None

remove-tuple
Input: <<Relation>>, Cluster ID
Output: None

union
Input: <<Relation>>, <<Relation>>, result_name
Output:<<Temp Relation>>

selection
Input: <<Relation>>,list of the form
(attribute, operator, value), result_name
Output: <<Temp Relation>>

projection
Input: <<Relation>>, list of attributes, result_nam
Output: <<Temp Relation>>

difference
Input: <<Relation>>, <<Relation>>, result_name
Output: <<Temp Relation>>

Cartesian product
Input:<<Relation>>, <<Relation>>, result_name
Output: <<Temp Relation>>

verify union compatibility
Input: <<Relation>>, <<Relation>>
Output: <<Relation>>, <<Relation>>

remove duplicates and write R2

81

▽ Temp Relation

NULL
▽

relation   name

NULL
▽

attribute-names

NULL
▽

attr-types

▧ Temp Relation

▧
union

Input: <<Temp Relation>>, <<Temp Relation>>, result_name
Output:<<Temp Relation>>

▧
selection

Input: <<Relation>>,list of the form
(attribute, operator, value), result_name
Output: <<Temp Relation>>

▧
projection

Input: <<Temp Relation>>, list of attributes, result_name
Output: <<Temp Relation>>

▧
difference

Input: <<Temp Relation>>, <<Temp Relation>>, result_name
Output: <<Temp Relation>>

▧
Cartesian   product

Input:<<Temp Relation>>, <<Temp Relation>>, result_name
Output: <<Temp Relation>>

▧
decompose   lists

Input: an empty list, the list to be decomposed
Output: decomposed list

▧
open Temp Relation

Input: an empty list, the list to be decomposed
Output: decomposed list

82

▽ Record

attr-types

attr-names

keys

▣ Record

Inputs: <<Record>>
Output: <<Record>> with value of attr-types set to a list of attribute values

attr-types

Input: <<Record>>
Output: <<Record>> with keys set equal to a list of all attribute names.
All attributes will be treated as key values unless this method is over-shadowed

keys

Input: <<Record>>
Output: <<Record>> with attr-names set to a list of attribute names

attr-names

Input: <<Record>>
Output: none
Uses display primitive to show an instance of a Record. This method should be overshadowed by the user of these classes

display-yourself

Input: <<Record>>
Output: primary key value
This method selects the first attribute created for objects of this class and returns its value as the primary key. This Method should be OVER-SHADOWED by the user

get primary key

Input: 2 tuples
Output: None. Suceeds or Fails
Compares all attributes of two tuples and determines equality.

=

Input: 2 tuples
Output: None. Suceeds or Fails
Compares all attributes of two tuples and determines inequality.

≠

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines <.

<

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines >.

>

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines ≤.

≤

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines ≥.

≥

APPENDIX C - SOURCE CODE

⊕Classes

System

Application Menu Menu Item Window Window Item

Database Relation Record

Temp Relation

▽ Database

$$..$$
$$\nabla$$

flle   name

$$0$$
$$\nabla$$

flle   volume   ld

$$0$$
$$\nabla$$

database   ld

$$0$$
$$\nabla$$

Structure   Table   ld

$$()$$
$$\nabla$$

temp   relation   llst

Database

new-db   Input: None
Output: DBId

delete-db   Input: None
Output: None

new-relation   Input: <<Database>>
Output: <<Relation>>

open-db   Input: None
Output: DBId

rename-db   Input: None
Output: None

delete-relation   Input: None
Output: None

close-db   Input: None
Output: None

display-yourself   Input: None
Output: None
Uses the show primitive
to display a list containing
the names of all relations
in the Database

temp   relation   llst

§1. New Database Name:

§1. No file was created, would you like to try again ?

▨Database/new-db 1:2create table & key 1:1

§1

table-new

§3

§2

table-close `ccccc` key-new
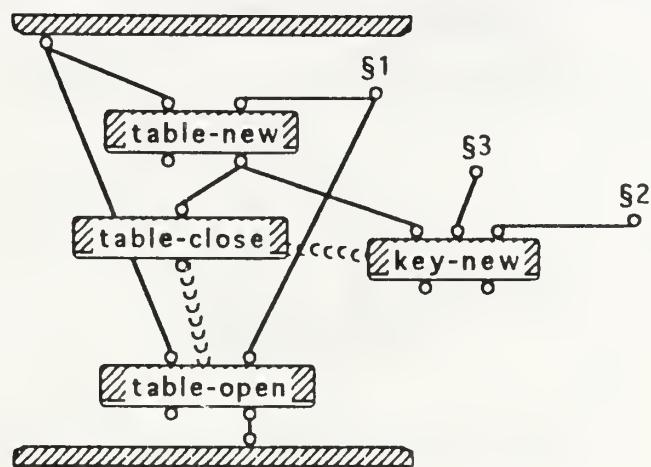
table-open

1. Structure Table
2. (string sensitive unique)
3. Relation Table

▨Database/new-db 1:2make instance 1:1

Database

file name

file volume Id

database Id

Structure Table Id

db-shutdown >>>>>>>>>>>>: ('sFdB')

get-file

NULL ✓

select a mode

db-open

0 ⊗

make DB instance

current DB

DBId

§1

No   Yes

answer

Yes ☒ >>>> //open-db

DBId

§1. No database was selected, would you like to try again ?

NULL

DBId
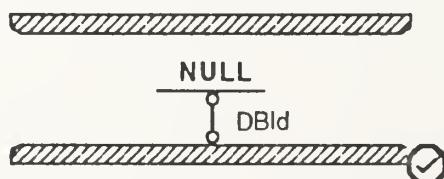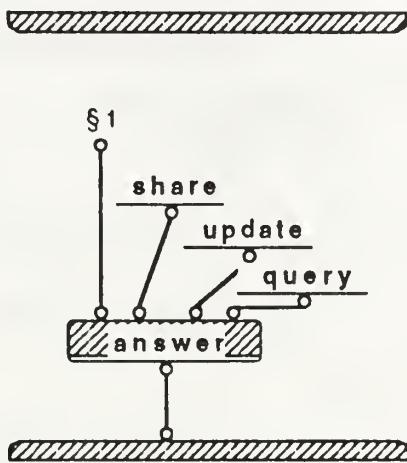
. What mode do you want to open the database in?
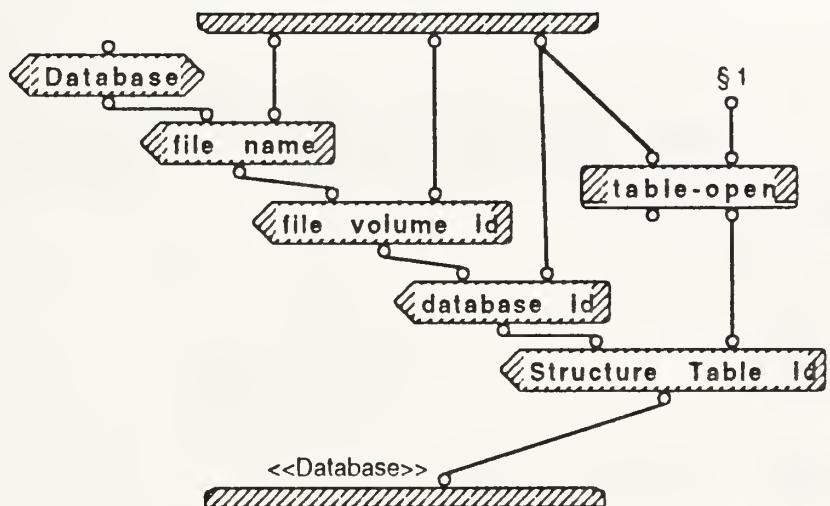
. Structure Table

Database/close-db 1:1open file 1:1
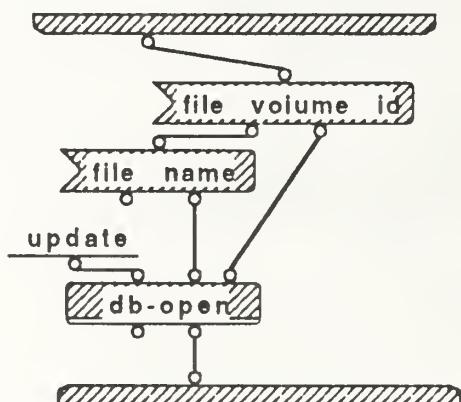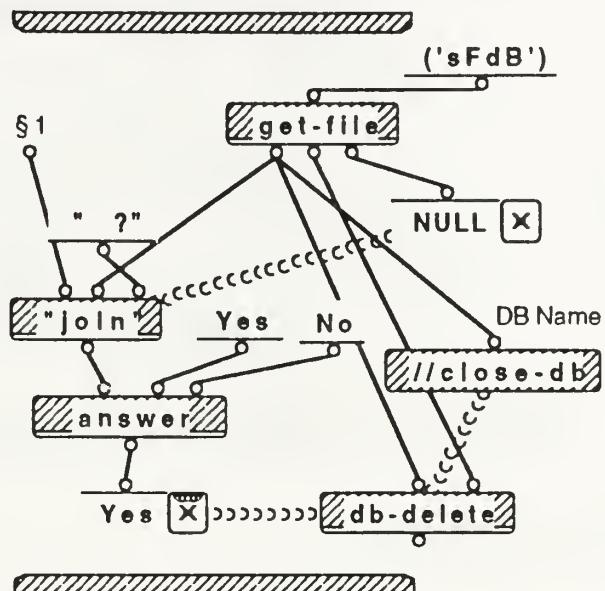
. "Are you sure you want to delete the database files associated with the database: "

. No database was selected, would you like to try again ?

§1. Which database do you want to rename ?
§2. What is the new name for the database ?

§1. No datbase was selected, would you like to try again ?

92

1. Relation Table

<<Database>>

database Id

table-lis

get class name ✕

make Relation inst.

keys

store relation in ST ⊗

~primary   key

attach-r

add Relation to DB ⊗

§1

No   Yes

answer

Yes ✕)>>>> //new

§1. No Relation selected, would you like to try again?

List of tables
that already   (Record)   ()
exist in DB

make class list

detach-l   removes "Record"
             from the list

(...)
make eligible list

§1
select

NULL ⊘

Selected Class name

1. Please select the class being stored in this relation.
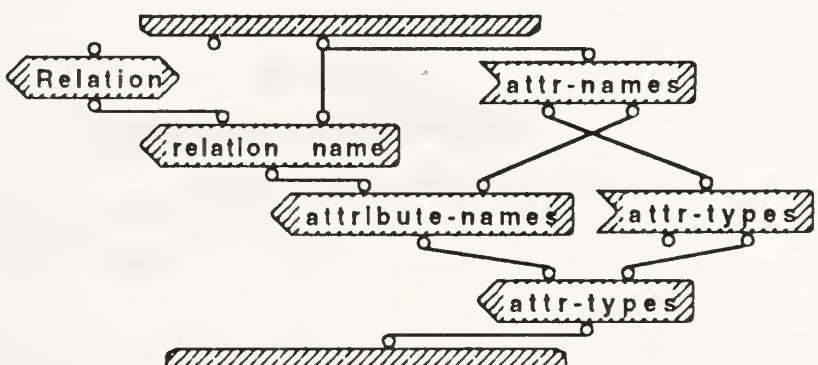
Relation   attr-names

relation name

attribute-names   attr-types

attr-types
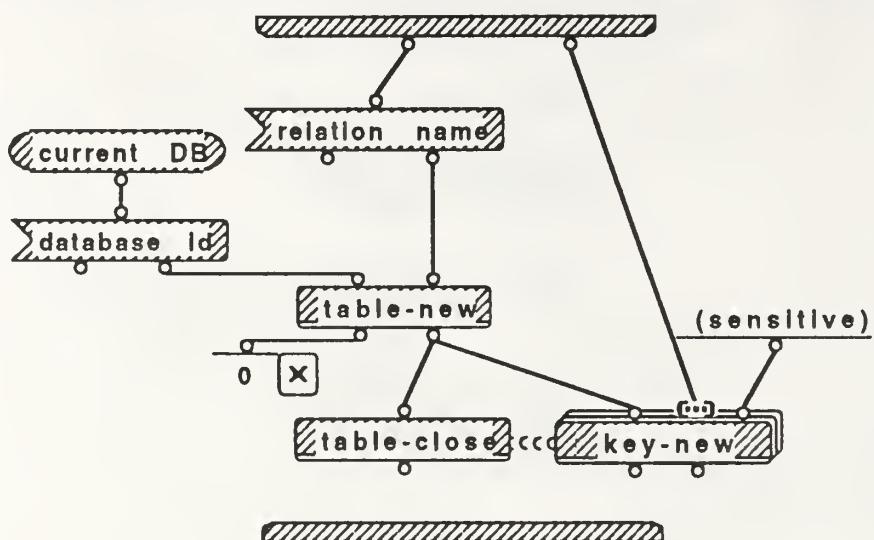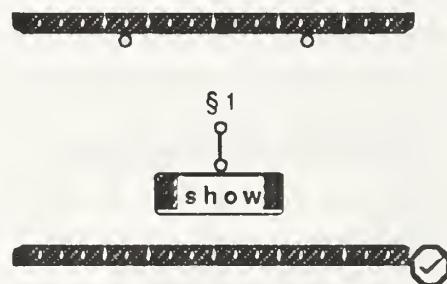
§1. The database is not opened in the correct mode. Must be in update mode for this to succeed.

. Structure Table

. You must be in update mode to perform this operation.

(in)

detach-nth
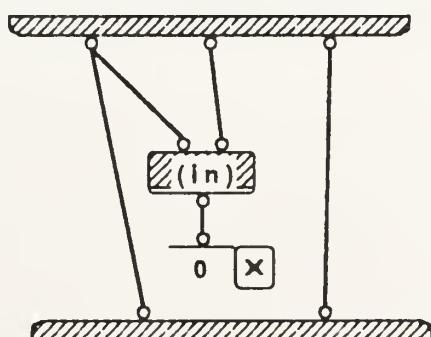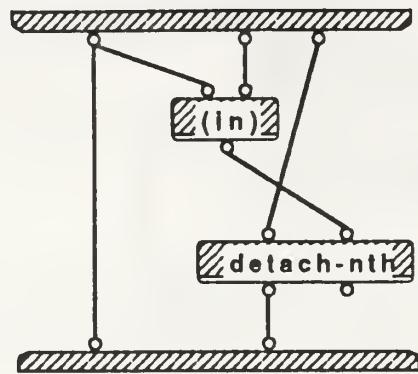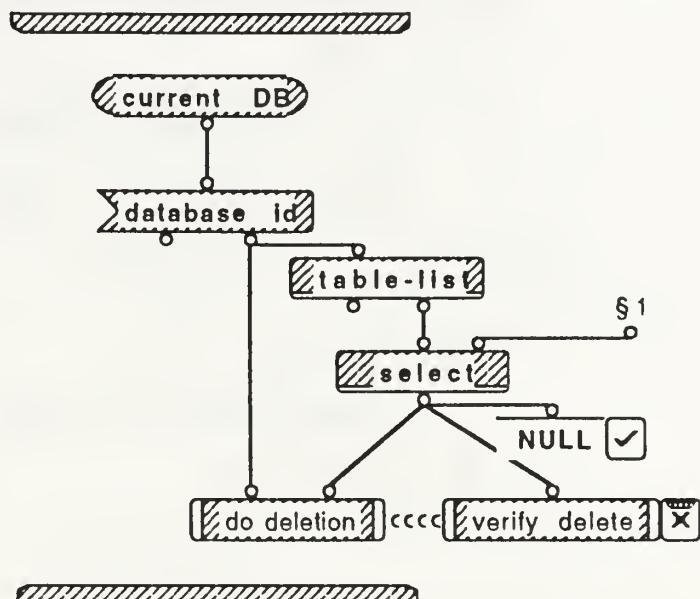
. "Select the table to be deleted.

. No table was selected, would you like to try again ?
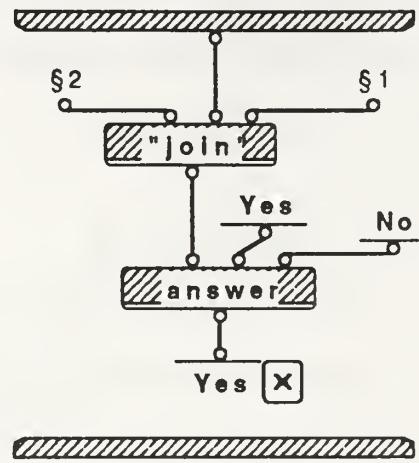
§1. " and all the keys and data associated with it ?"
§2. "Are You sure you want to delete the table named "
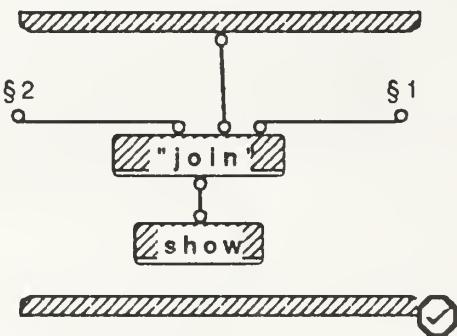
§1. Will not be deleted!
§2. "The Relation "
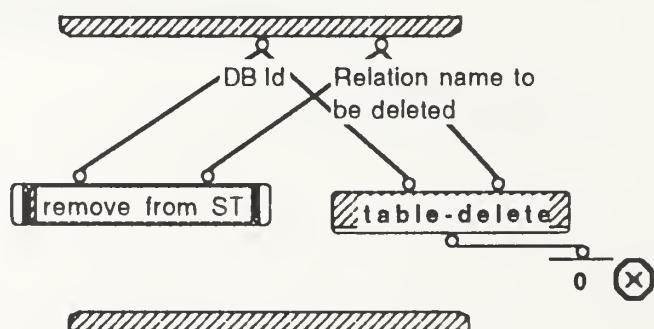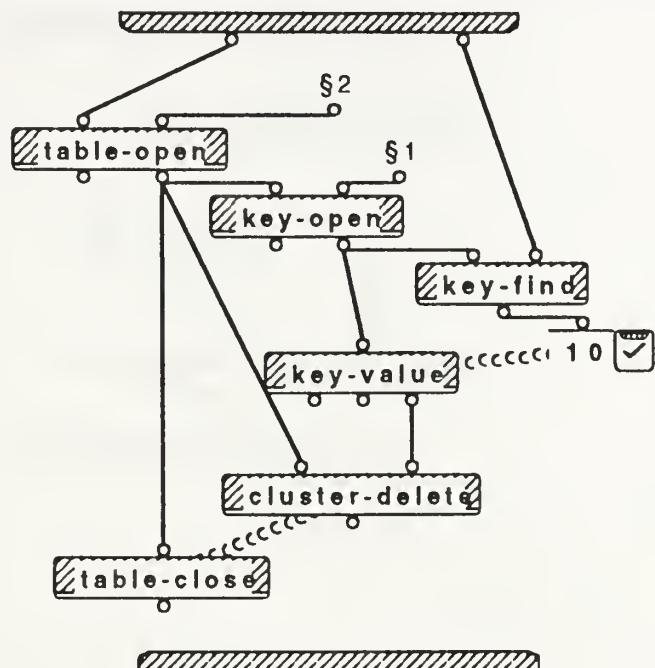
1. Relation name
2. Structure Table
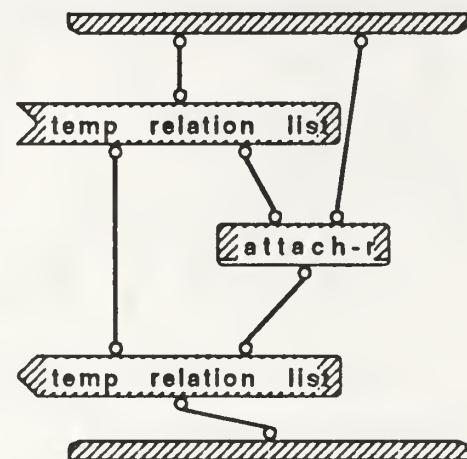
∇ Record

attr-types

attr-names

keys

🗂 Record

Inputs: <<Record>>
Output: <<Record>> with value of attr-types set to a list of attribute values

attr-types

Input: <<Record>>
Output: none
Uses display primitive to show an instance of a Record. This method should be overshadowed by the user of these classes

display-yourself

Input: <<Record>>
Output: <<Record>> with keyys set equal to a list of all attribute names.
All attributes will be treated as key values unless this method is over-shadowed

keys

Input: <<Record>>
Output: primary key value
This method selects the first attribute created for objects of this class and returns its value as the primary key. This Method should be OVER-SHADOWED by the user

get primary key

Input: <<Record>>
Output: <<Record>> with attr-names set to a list of attribute names

attr-names

Input: 2 tuples
Output: None. Suceeds or Fails
Compares all attributes of two tuples and determines equality.

=

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines >.

>

Input: 2 tuples
Output: None. Suceeds or Fails
Compares all attributes of two tuples and determines inequality.

≠

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines ≤.

≤

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines <.

<

Input: 2 tuples
Output: None. Suceeds or Fails
Compares first attributes of each tuple and determines ≥.

≥

105

tuple 1   tuple 2

tuple 1   tuple 2

tuple 1   tuple 2

:

tuple 1   tuple 2

attr-names   attr-names

detach-   detach-

≤ ×

tuple 1   tuple 2

111

▽ Relation

```
          ..
          ▽
    relation   name

         NULL
          ▽
    attribute-names

          ( )
          ▽
      attr-types
```

▦Relation

open table
Input: <<Relation>>
Output: TableId

open to first tuple
Input: <<Relation>>
Output: Primary key Id, Table Id

select-relation
Input: DBId
Output: TableId

display-yourself
Input: <<Relation>>
Output: None

make Temp Relation
Input: name, <<Relation>>
Output: <<Temp Relation>>

add-tuple
Input: Table Id, tuple
Output: None

remove-tuple
Input: <<Relation>>, Cluster ID
Output: None

union
Input: <<Relation>>, <<Relation>>, result_name
Output:<<Temp Relation>>

selection
Input: <<Relation>>,list of the form
(attribute, operator, value), result_name
Output: <<Temp Relation>>

projection
Input: <<Relation>>, list of attributes, result_name
Output: <<Temp Relation>>

difference
Input: <<Relation>>, <<Relation>>, result_name
Output: <<Temp Relation>>

Cartesian product
Input:<<Relation>>, <<Relation>>, result_name
Output: <<Temp Relation>>

verify union compatibility
Input: <<Relation>>, <<Relation>>
Output: <<Relation>>, <<Relation>>

remove duplicates and write R2

113

§1. Select the Relation you wish to open.
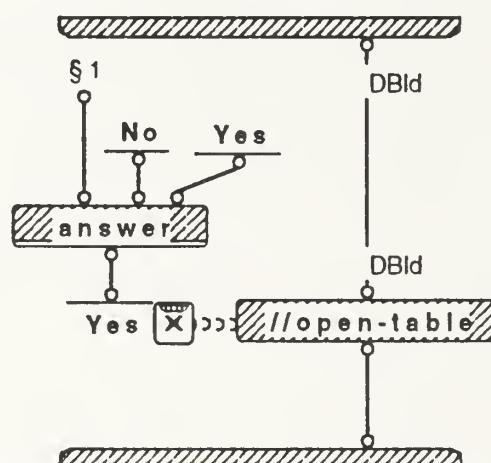
§1. No table was selected, would you like to try again ?

<<Relation>>

//open table

~primary key

key-open

key-firs

table-close ccccc do all records ccccccccc 0 ⊗

key-read

0 ☒ >>>>>>>>>>>>>>>>> key-next

display

117

current DB

User selected
Relation name

Relation 1

/temp relation list

Temp Relation

relation name

attr-types

attr-types

attribute-names

attribute-names

Check to see if
there are any keys
associated with this
table. If not return NULL

key list

NULL

<<Relation 1>> <<Relation 2>>　　result name

//verify　union　compatibili☒　>>>>>　✍make Temp Relation✍

<<Temp Relation>>

✍open Temp Relation✍

make R2 list　　write R1 to Temp R✍

Temp
Relation

List of every
tuple in R1

✍/open　table✍

//remove　duplicates　and　write R✍

▨Relation/union 2:2

<<Relation 1>>　<<Relation 2>>

§ 1

✍show✍

NULL

<<Temp Relation>>

The two relations are not union compatible

current DB

temp relation list

User selected
Relation name

Relation 1

Temp Relation

relation name

attr-types

attr-types

attribute-names

attribute-names

<<Temp Relation>>

relation name

create temp relation table >>>> /open table

~primary key

key-open

//open to first tuple

( )

read & write

table-close

Temp
Relation

List of every
instance in R

//open to first tuple

( )

make list

table-close

current DB

database id

table-new

~primary key

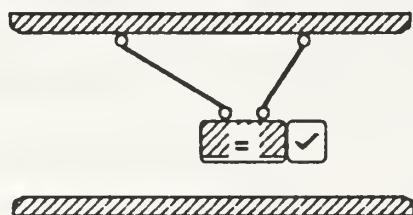table-close <<<<<<<<< key-new

key-read

0

key-next

attach-r

("1")

cluster-write

Relation/union 1:2make R2 list 1:1make list 1:2

Relation/union 1:2make R2 list 1:1make list 2:2

Relation/union 1:2make R2 list 1:1make list 1:2do all instances 1:1

128

key-read

0 X

key-next

inst-to-lis

("1")

cluster-write <<<<<<<<<<<<<<< X

Relation/selection 1:1get the tuples 2:2
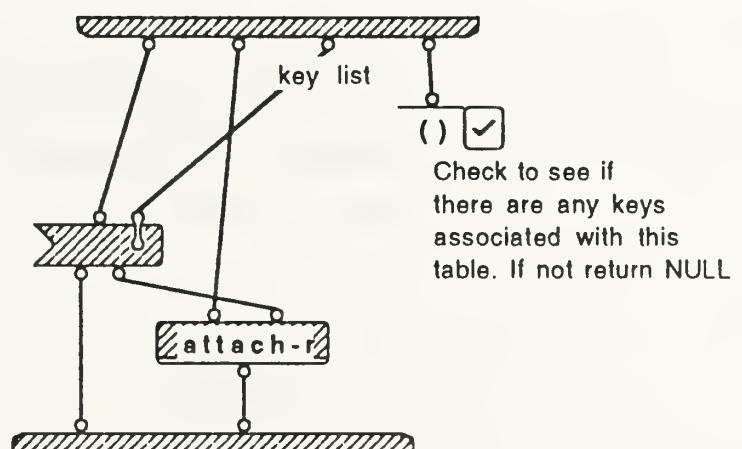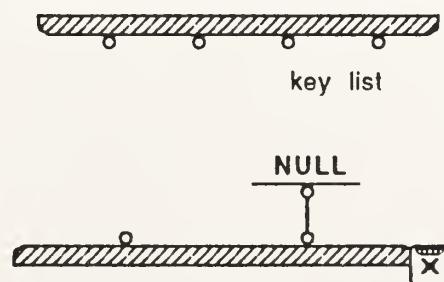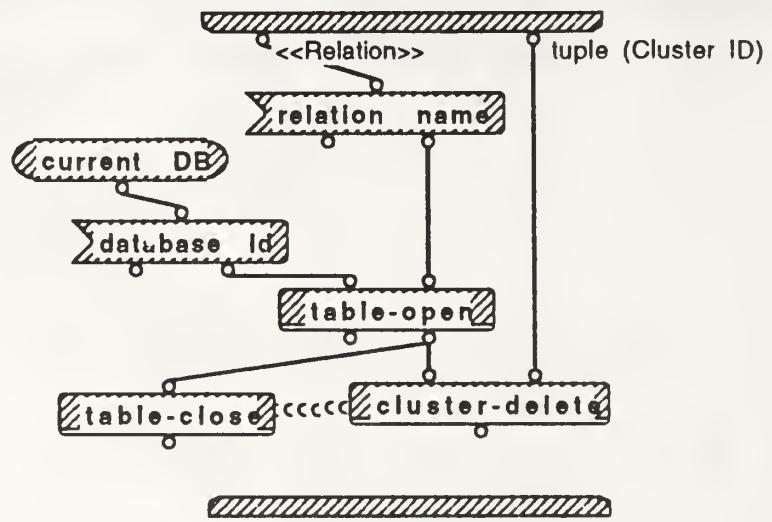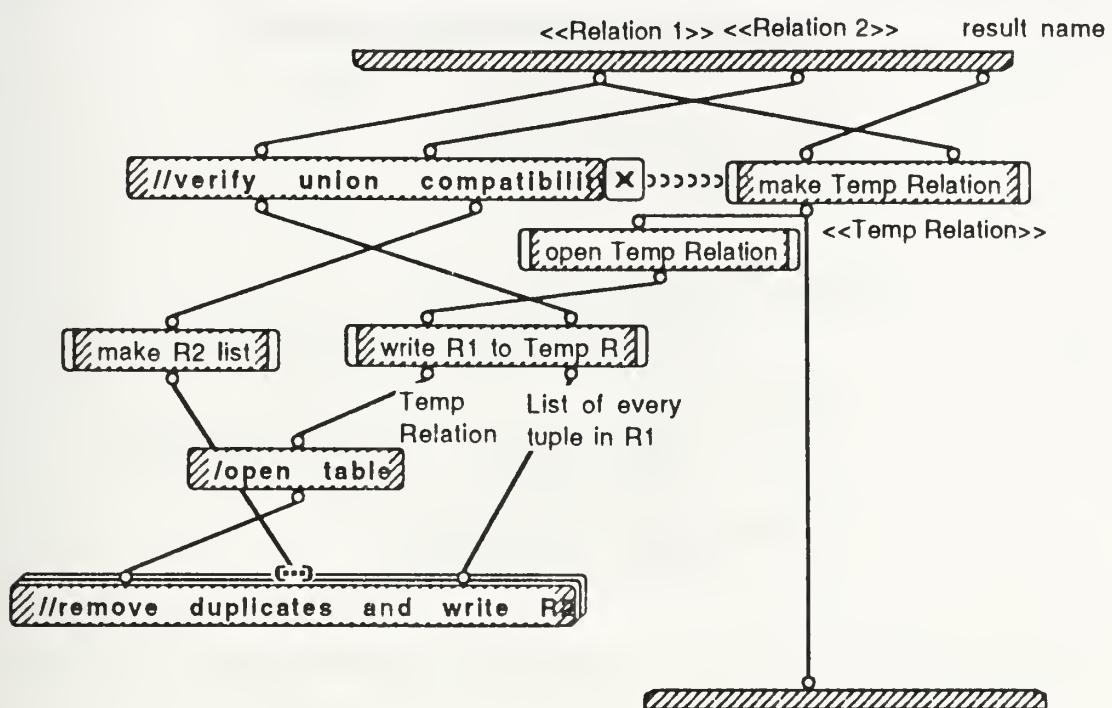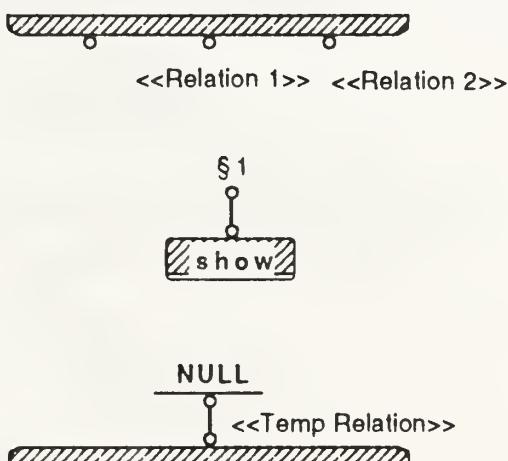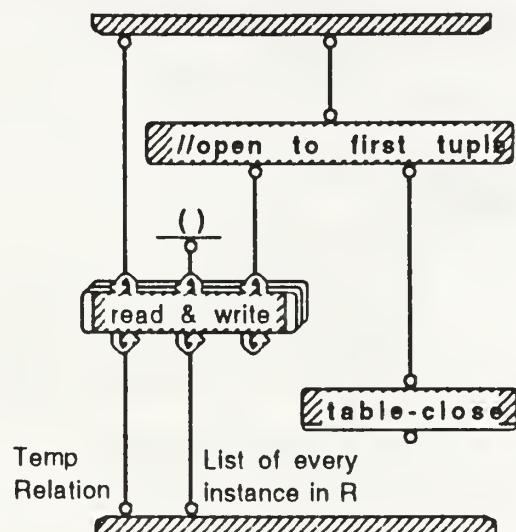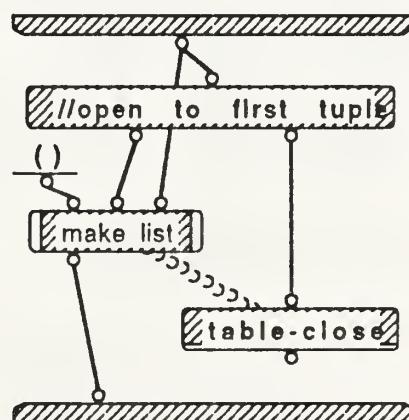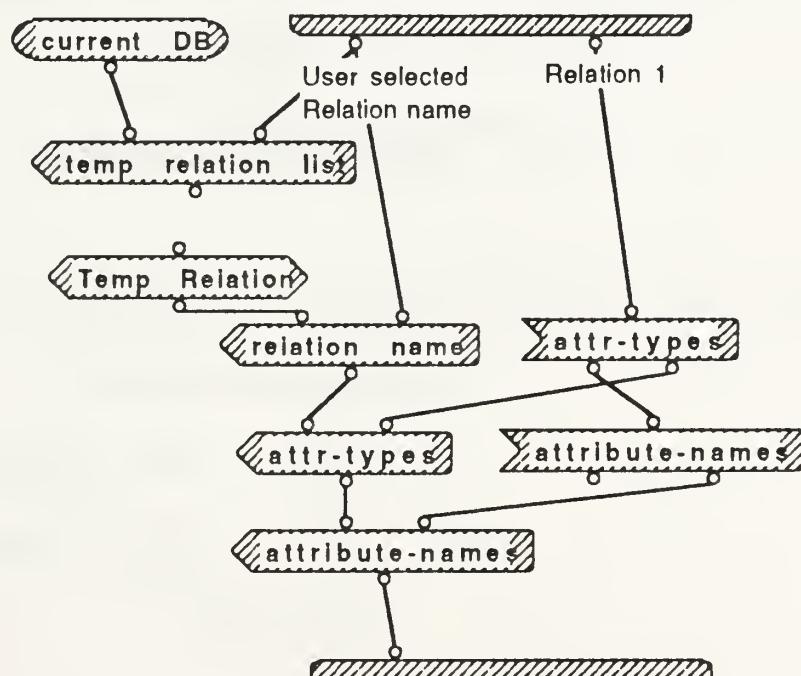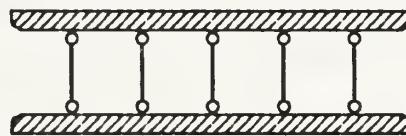
Relation/selection 1:1open Temp Relation 1:1create temp relation table 1:1
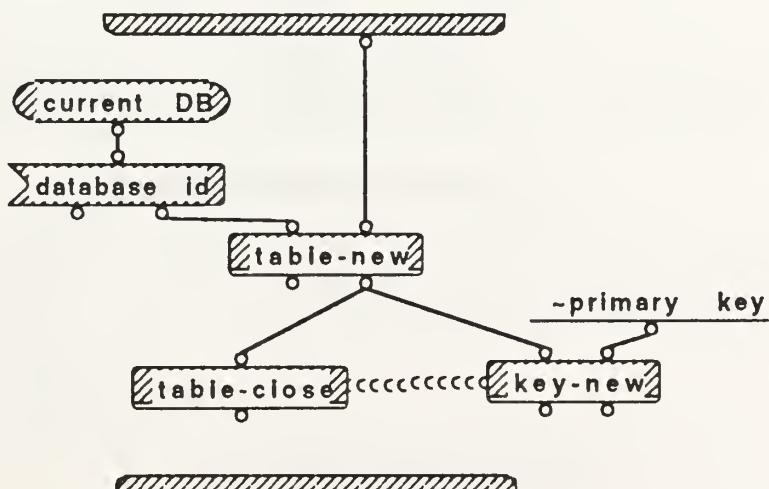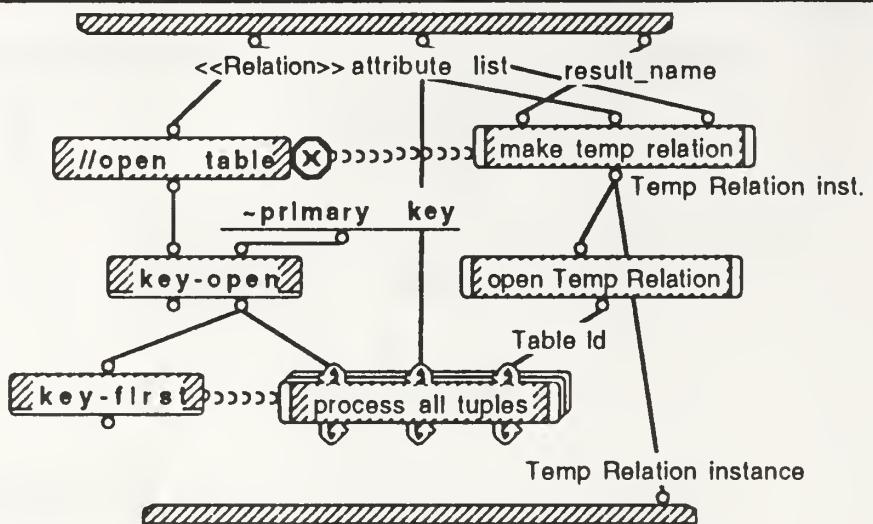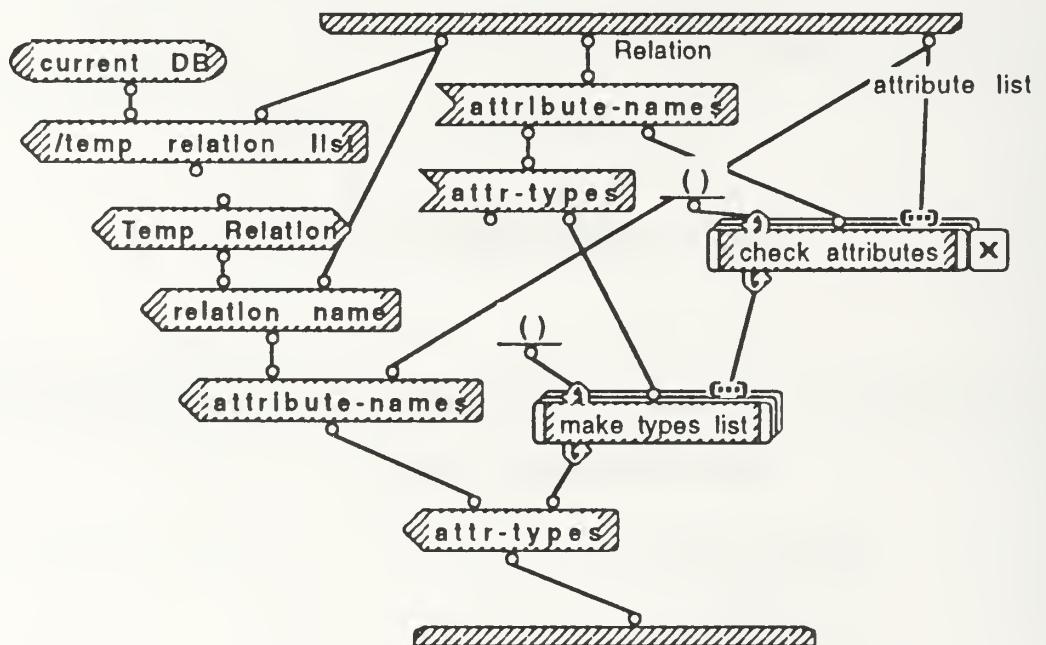
current DB

database id

table-new

~primary key

table-close <<<<<<<<<< key-new

Relation/projection 1:1 make temp relation 2:2

Relation attribute list

§1

show
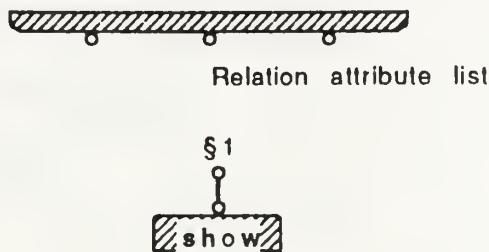
1. One of the attributes selected does not exist in the selected Relation

Relation/projection 1:1 open Temp Relation 1:1

<<Temp Relation>>

relation name

create temp relation table >>> //open table

~primary key

key-open

Relation/projection 1:1 process all tuples 1:1

key-read

attribute list

Table id

0 X >>>>>>>>>>>>>>>>>>> write values to Temp Relation

key-next

§1. does not exist in the specified relation.
§2. The attribute:

current DB

database id

table-new

~primary key

table-close ⊂⊂⊂⊂⊂⊂⊂⊂⊂ key-new

( )

attribute list

get all attributes

("1")

cluster-write

attribute list

attach-r

134
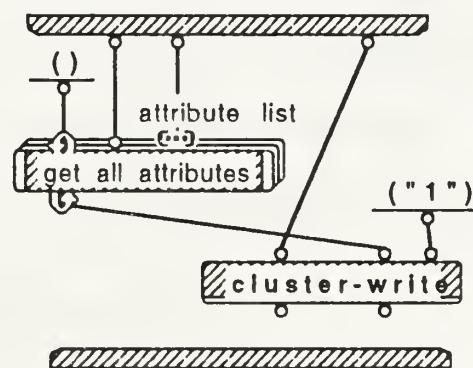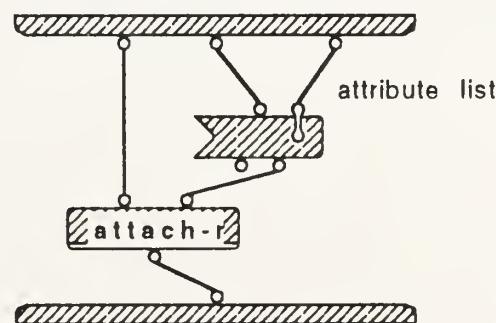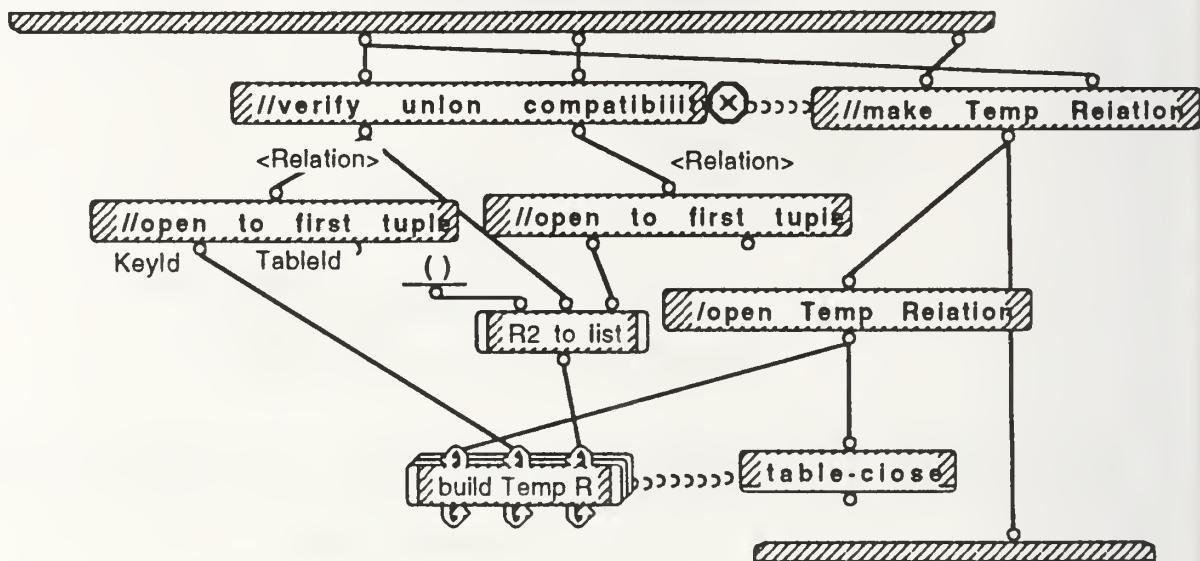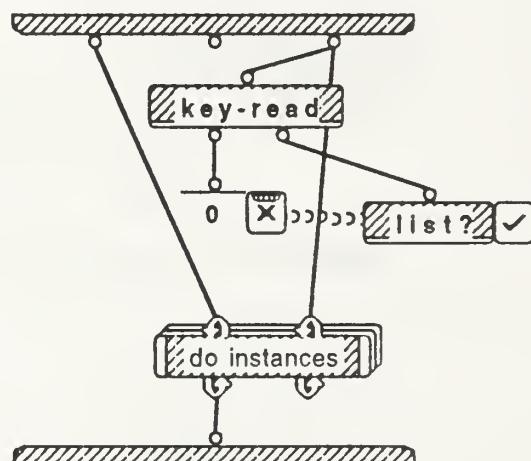
⧄Relation/difference 1:1R2 to list 2:2

⧄Relation/difference 1:1build Temp R 1:2

if there is a tuple in
R2 that equals the current
tuple, this methd fails

⧄Relation/difference 1:1build Temp R 2:2

136

Relation/Cartesian product 1:1

<<Relation>>   <<Relation>>   result_name

//open table   make Temp relation   <<Relation>>

~primary key   //open table

key-open   <<Temp Relation>>   ~primary key

key-first   /open Temp Relation   key-open

process tuples

Relation/Cartesian product 1:1make Temp relation 1:1
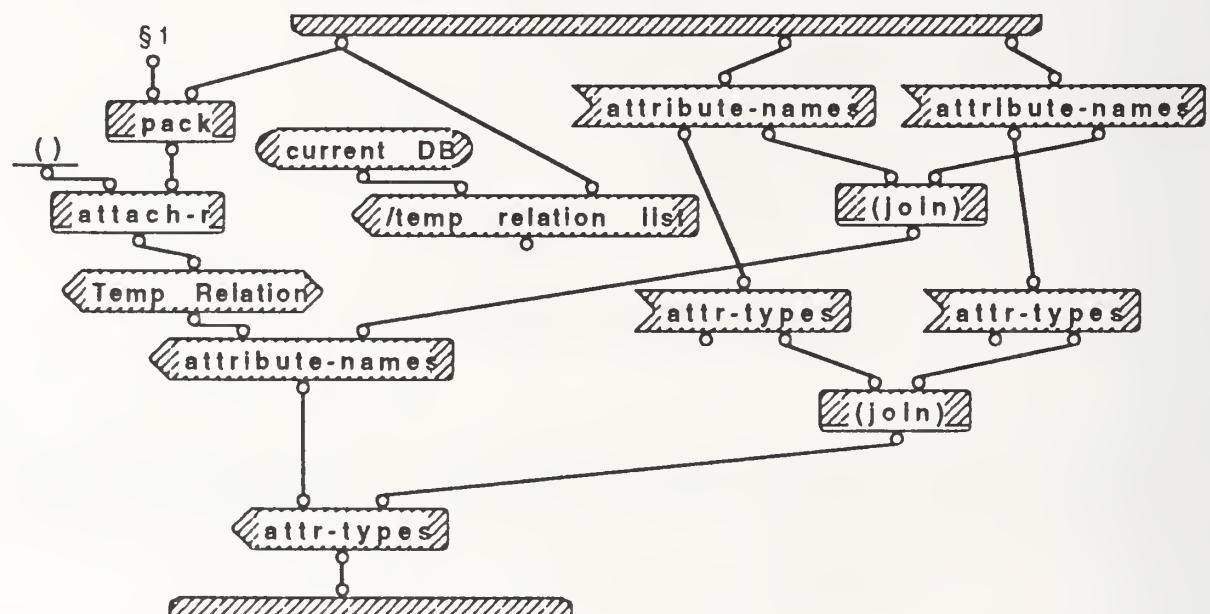
§1
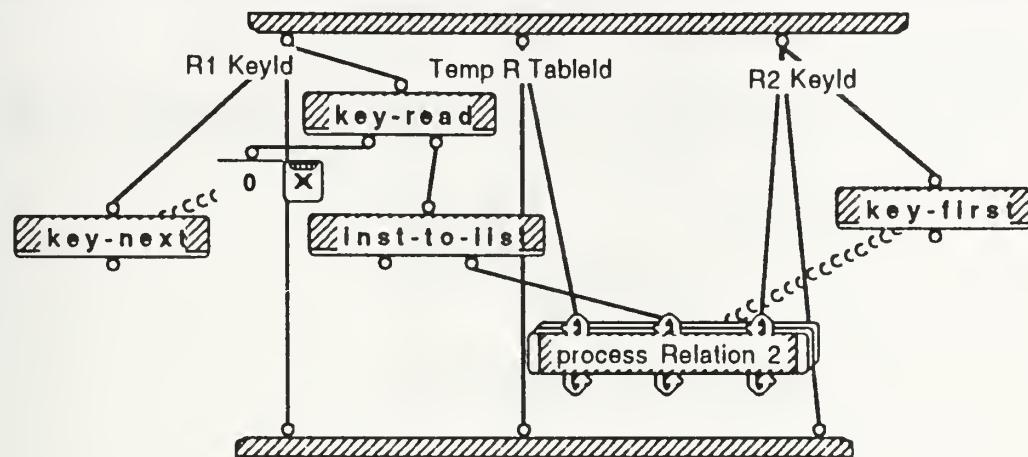
pack   attribute-names   attribute-names

( )   current DB

attach-r   /temp relation list   (join)

Temp Relation   attr-types   attr-types

attribute-names   (join)

attr-types
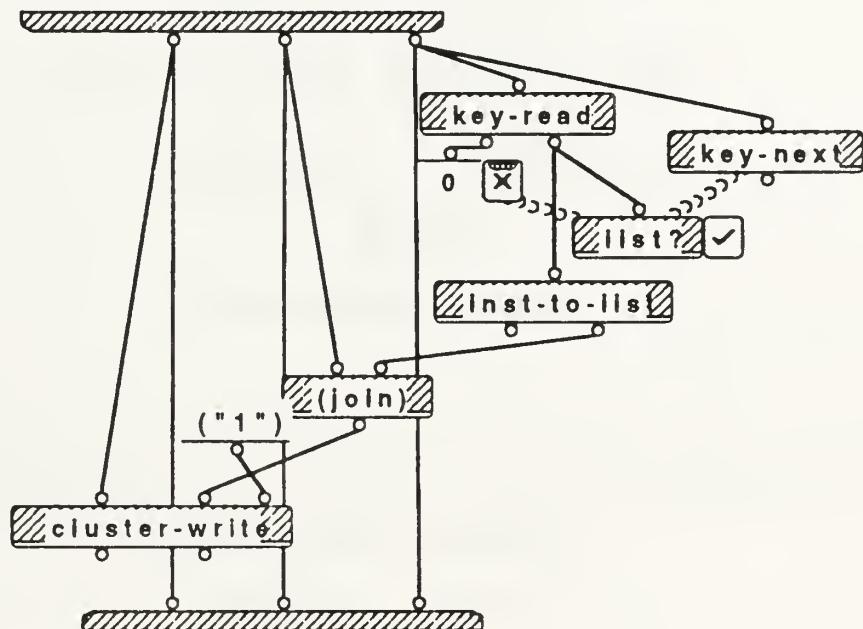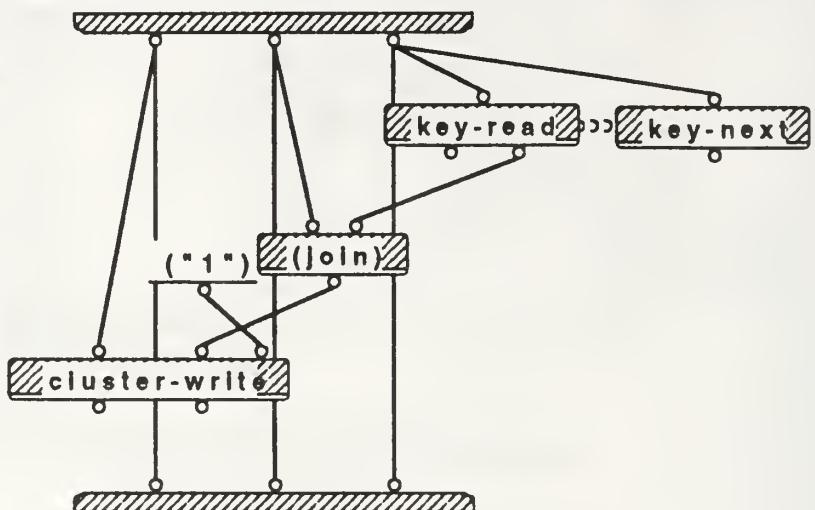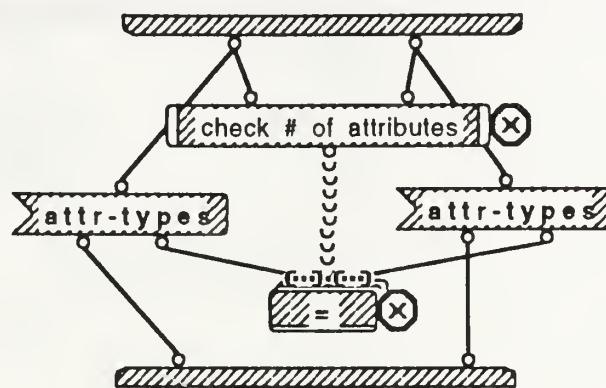
§1. relation name

1. Relations do not have the same number of attributes, and cannot be unioned together.

Relation/remove duplicates and write R2 1:2
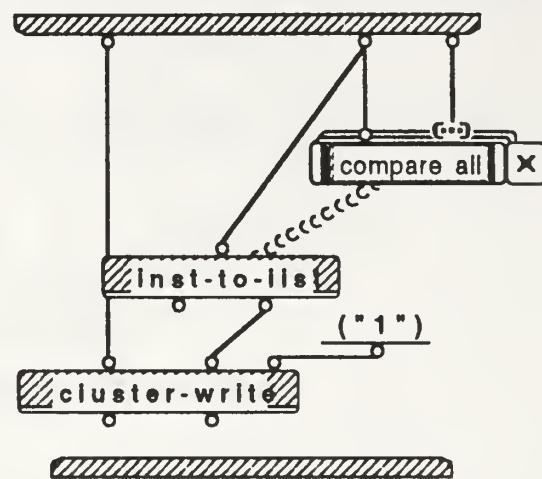
Relation/remove duplicates and write R2 2:2

Relation/remove duplicates and write R2 1:2compare all 1:2

Relation/remove duplicates and write R2 1:2compare all 2:2

142

▽ Temp Relation

NULL
▼
relation name

NULL
▼
attribute-names

NULL
▼
attr-types

🔳 Temp Relation

union
Input: <<Temp Relation>>, <<Temp Relation>>, result_name
Output:<<Temp Relation>>

selection
Input: <<Relation>>,list of the form
(attribute, operator, value), result_name
Output: <<Temp Relation>>

projection
Input: <<Temp Relation>>, list of attributes, result_name
Output: <<Temp Relation>>

difference
Input: <<Temp Relation>>, <<Temp Relation>>, result_name
Output: <<Temp Relation>>

Cartesian   product
Input:<<Temp Relation>>, <<Temp Relation>>, result_name
Output: <<Temp Relation>>

decompose   lists
Input: an empty list, the list to be decomposed
Output: decomposed list

open Temp Relation
Input: an empty list, the list to be decomposed
Output: decomposed list

<<Relation 1>> <<Relation 2>>   result name

//verify   union   compatibili[X]>>> //make Temp Relation

//open Temp Relation

make R2 list   write R1 to Temp R

( )   ( )

//decompose   lists   //decompose   lists

Temp   List of every
Relation   tuple in R1

(...)   remove duplicates from R2 >>>>>>>>>> table-close

<<Relation<<Relation 2>>

§1

//show

NULL

<<Temp Relation>>

§1. The two relations are not union compatible

<<Relation 1>> <<Relation 2>>   result name

//verify union compatibili[X] >>> //make Temp Relation

//open Temp Relation

make R2 list   write R1 to Temp R

()   ( )

//decompose lists   //decompose lists

Temp
Relation

List of every
tuple in R1

remove duplicates from R2 >>>>>>>>> table-close

<<Relatior <<Relation 2>>

§1

show

NULL

<<Temp Relation>>

§1. The two relations are not union compatible

145

Temp Relation

List of every instance in R

146

Temp Relation/union 1:2remove duplicates from R2 1:2

Temp Relation/union 1:2remove duplicates from R2 2:2

Temp Relation/union 1:2make R2 list 1:1make list 1:2

150

Temp Relation/projection 1:1make temp relation 2:2

Relation attribute list

§1

show
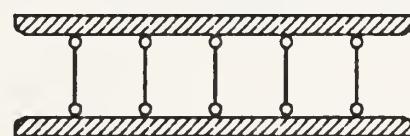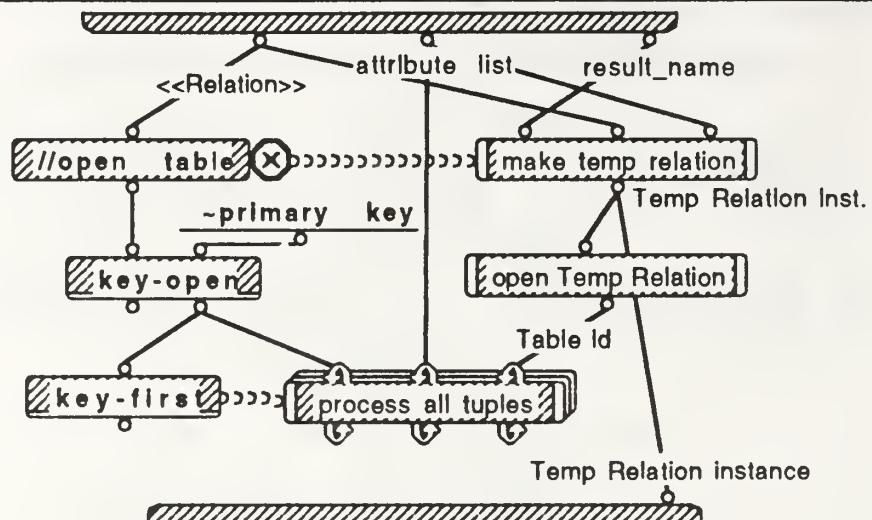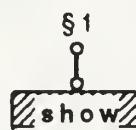
§1. One of the attributes selected does not exist in the selected Relation

Temp Relation/projection 1:1open Temp Relation 1:1

<<Temp Relation>>

relation name

create temp relation table >>> //open table

~primary key

key-open

Temp Relation/projection 1:1process all tuples 1:1

key-read

attribute list

Table id

0 X >>>>>>>>>>>>>>>>>>> write values to Temp Relation

key-next

§1. does not exist in the specified relation.
§2. The attribute:

current DB

database id

table-new

~primary key

table-close cccccccc key-new

( )

attribute list
(..)

get all attributes

("1")

cluster-write

attribute name

(in)

get-nth

attach-r

154
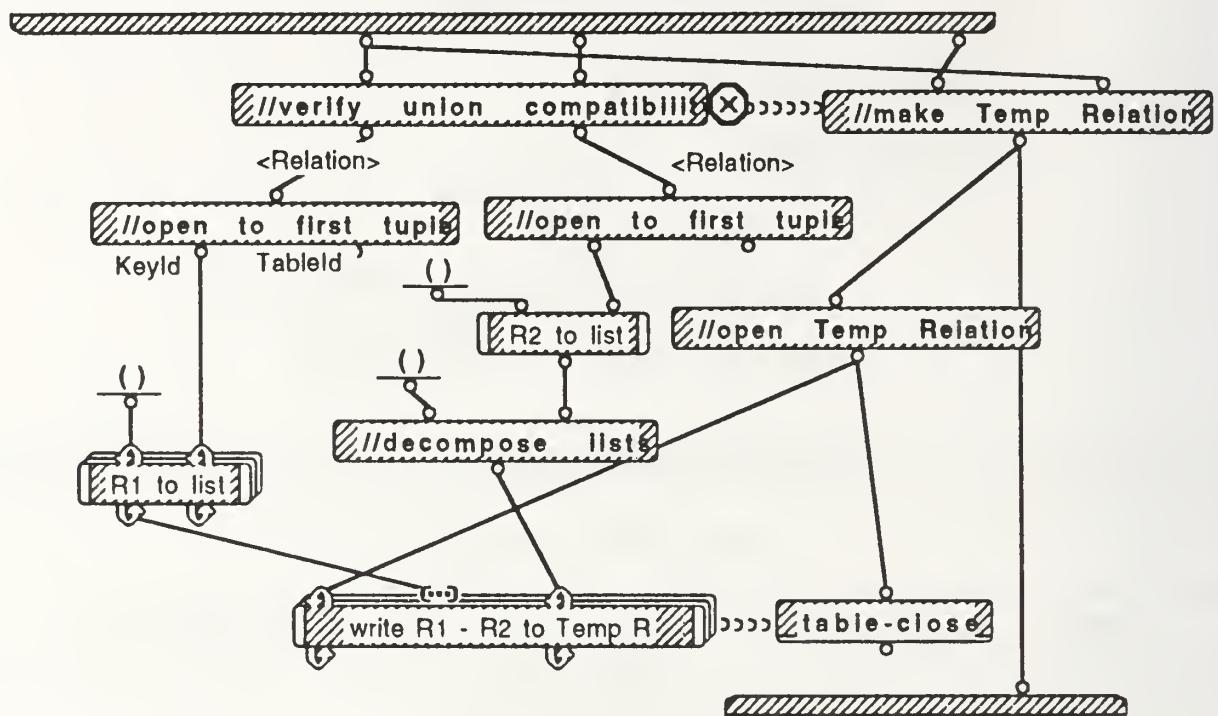
Temp Relation/difference 1:1R2 to list 2:2

Temp Relation/difference 1:1R1 to list 1:1
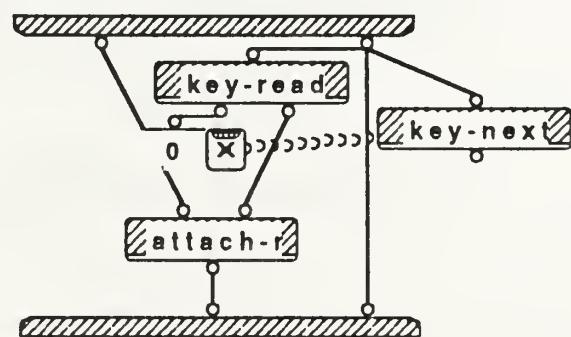
Temp Relation/difference 1:1write R1 – R2 to Temp R 1:2
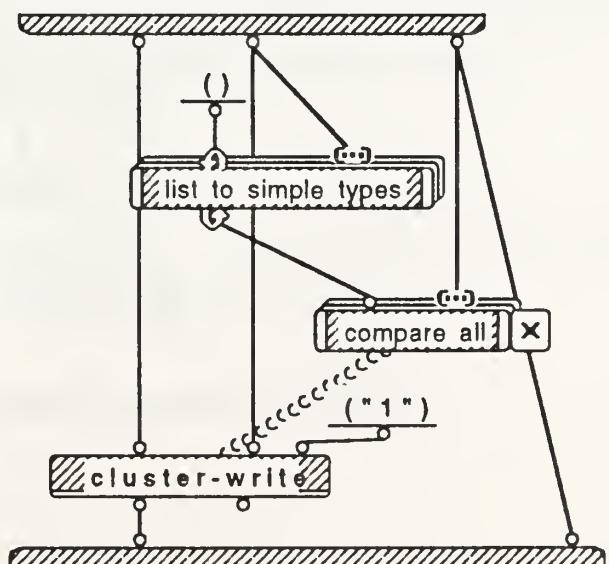
Temp Relation/difference 1:1write R1 - R2 to Temp R 2:2
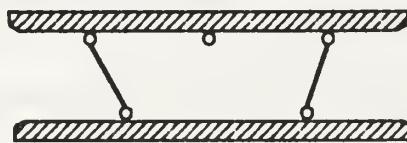
Temp Relation/difference 1:1R2 to list 1:2do instances 1:1
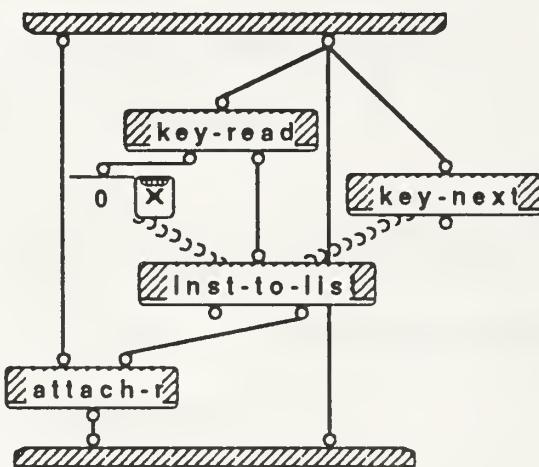
Temp Relation/difference 1:1R2 to list 2:2do list types 1:1

157

§1. relation name

158

R1 KeyId   Temp R TableId   R2 KeyId

key-read

0

key-next   process Relation 2   key-flrst
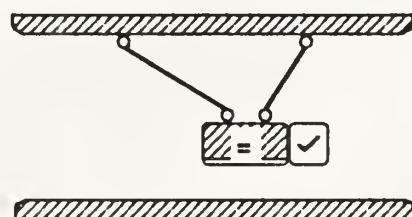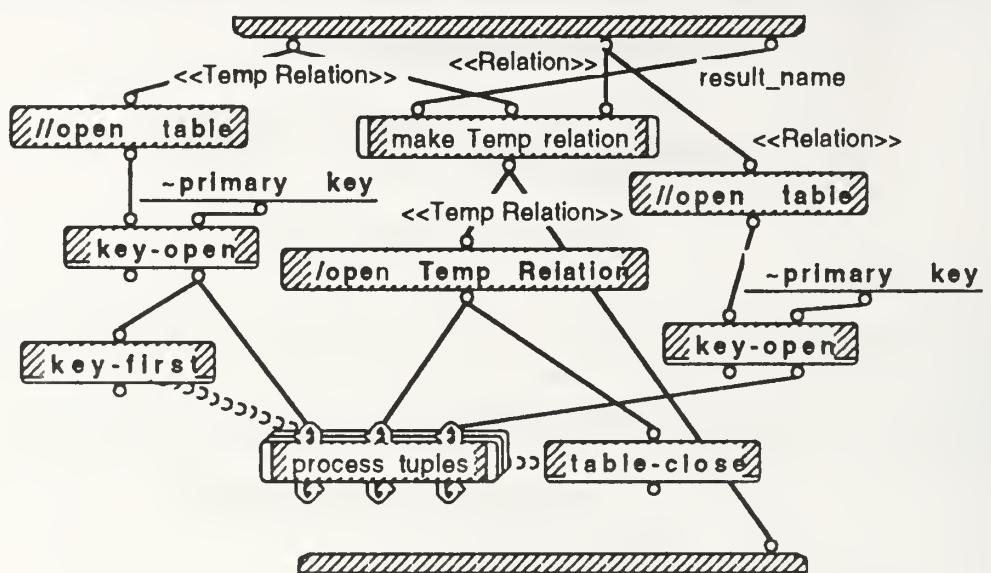
⬛Temp Relation/Cartesian product 1:1process tuples 1:1process Relation 2 1:2
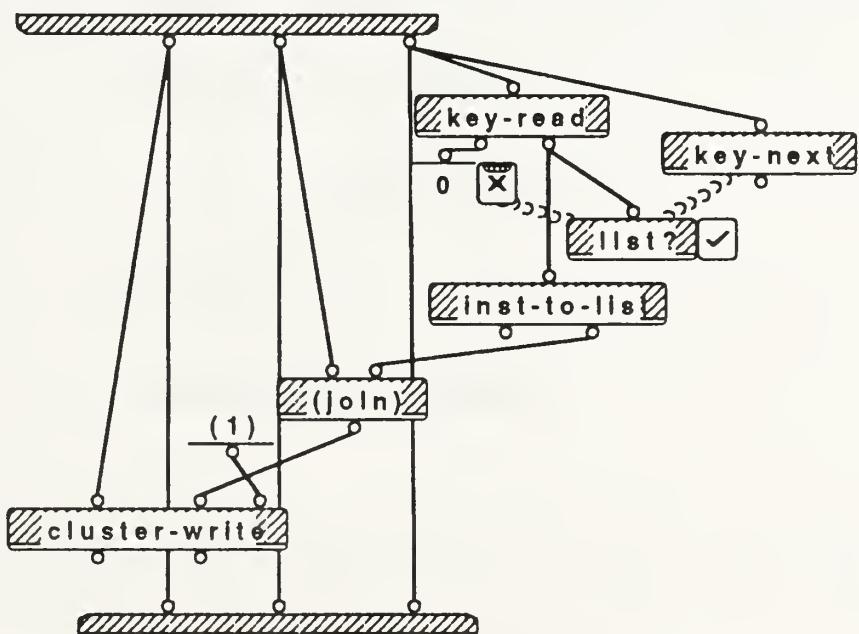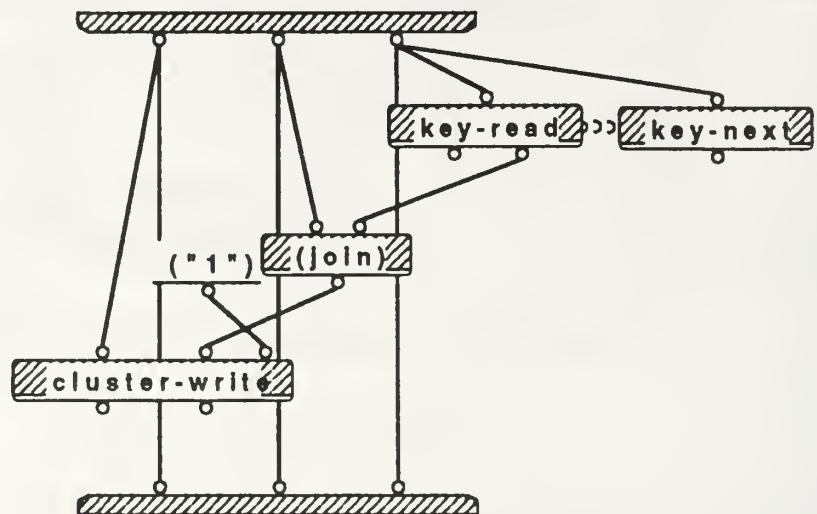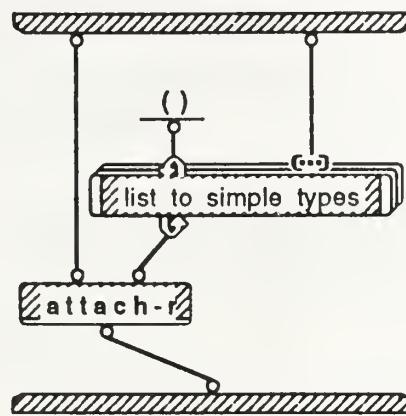
key-read

key-next

0

llst?

inst-to-lls

(join)

(1)

cluster-write
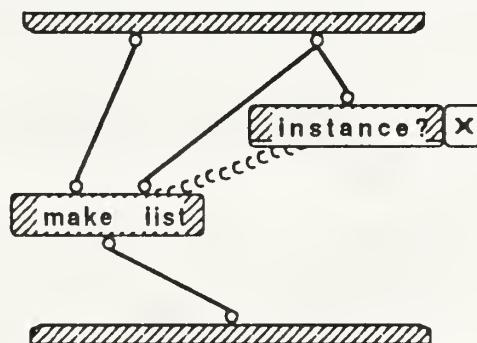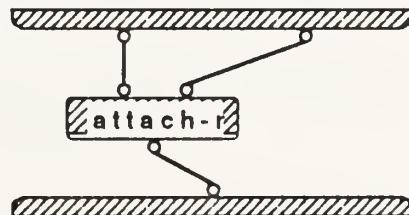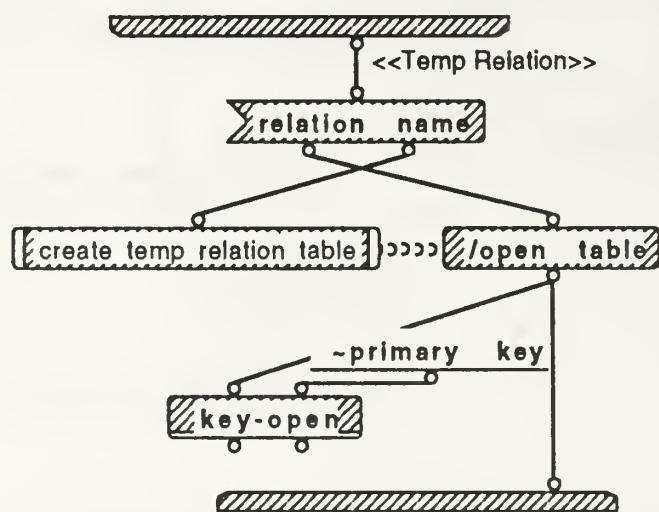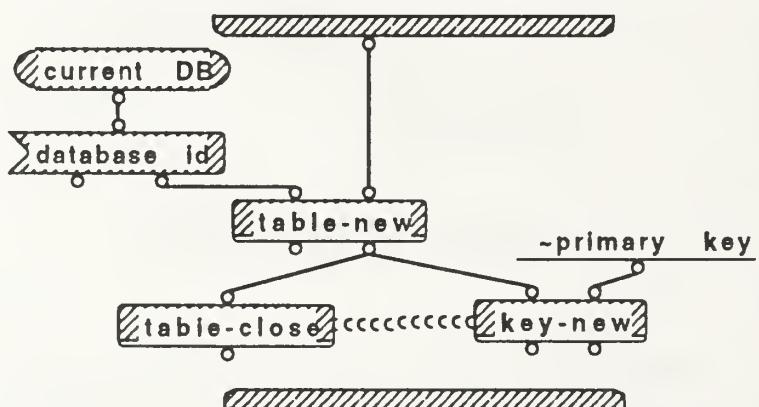
LIST OF REFERENCES

[Alag86]   Alagic, S., *Relational Database Technology*, Springer-Verlag, 1986.

[Booc91]   Booch, G., *Object-Oriented Design with Applications*, The Benjamin/ Cummings Publishing Company, Inc., 1991.

[Codd70]   Codd, E., "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, June 1970.

[EN89]   Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[FN92a]   Naval Postgraduate School Report NPSCS-92-005, *The Feasibility of Implementing a Relational/Object-Oriented Database Management System in the Gemstone Object-Oriented Database Management System,* by Filippi, S. C., and Nelson, M. L., April 1992 (draft).

[FN92b]   Naval Postgraduate School Report NPSCS-92-006, *The Feasibility of Implementing Conventional Database Models in an Object-Oriented Database Management System,* by Filippi, S. C., and Nelson, M. L., May 1992 (draft).

[Kim91]   Kim, W., "Object-oriented database systems: strengths and weaknesses", JOOP, July/August 1991.

[Meye88]   Meyer, B., *Object-oriented Software Construction*, 1988.

[Mica88]   Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming* v 1, April/May 1988.

[Nels88]   Nelson, M. L., *A Relational Object-Oriented Management System and An Encapsulateed Object-Oriented Prograsmming System,* Ph.D Dissertation, University of Central Florida, Orlando, Florida, December 1988.

[Nels90a]   Naval Postgraduate School Report NPS52-90-024, *An Introduction to Object-Oriented Programming,* by Michael L. Nelson, April 1990.

[Nels90b] Naval Postgraduate School Report NPS52-90-025, *Object-Oriented Database Management Systems*, by Michael L. Nelson, May 1990.

[NMO90] Nelson, M. L., Moshell, J. M., and Orooji, A., "A Relational Object-Oriented Management System", 9th Annual International Phoenix Conference on Computers and Communications (IPCCC '90) Proceedings, March 1990.

[NMO91] Nelson, M. L., Moshell, J. M., and Orooji, A., "The Case For Encapsulated Inheritance", *Proceedings of the 24th Annual Hawaii International Conference on System Sciences (HICSS-24), Vol II:Software Technology*, January 1991.

[Onto88] Ontologic Inc., VBase: For Object Applications, 1988.

[Onto90] Ontologic Inc., Ontos Object Database Documentation, Release 1.5, 1990.

[PK90] Perry, Dewayne E., and Kaiser, Gail E., "Adequate testing and Object-Oriented Programming", JOOP, January/February 1990.

[PBRV90] Premeriani, W. J., Blaha, M. R., Rumbaugh, J. E., and Varwig, T. A., "An Object-Oriented Relational Database", *Communications of the ACM*, November 1990.

[Serv89a] Servio Logic Development Corporation, Programming in OPAL, Part I, 1989.

[Serv89b] Servio Logic Development Corporation, Programming in OPAL, Part II, The OPAL Kernel Classes, 1989.

[Spea92] Spear, R., *A Relational Object-Oriented Database Management System*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1992 (draft).

[SB86] Stefik, M. and Bobrow, D. G., "Object-Oriented Programming : Themes and Variations",The AI Magazine, v 6, Winter 1986.

[TGS88a] The Gunakara Sun Systems, *Prograph Tutorial*, 1988.

[TGS88b] The Gunakara Sun Systems, *Prograph Reference*,1988.

[TGS91]   The Gunakara Sun Systems, *Prograph 2.5 Updates,* 1991.

[Wegn87]   Wegner, P. , "Dimensions of Object-Based Language Design", *OOPSLA '87 Conference Proceedings*, October 1987.

[Wu90]   Wu, C. T., "Development of a Visual Database Interface: An Object-Oriented Approach", *Application of Object-Oriented Programming* [PW90], 1990.

[ZM90]   Zdonick, S. B. and Maier D., "Fundamentals of Object-Oriented Databases", *Readings in Object-Oriented Database Systems*, 1990.

BIBLIOGRAPHY

[Beec88]   Beech, D., "Intensional Concepts in an Object Database Model", *OOPSLA Conference Proceedings*, September 1988.

[BZ87]   Bloom, T., and Zdonik, S. B., "Issues in the Design of Object-Oriented Database Programming Languages", *OOPSLA Conference Proceedings*, October 1987.

[KBCG87]   Kim, W., Banerjee, J., Chou, H., Garza, J. F., and Woelk, D., "Composite Object Support in an Object-Oriented Database System", *OOPSLA '87 Conference Proceedings*, October 1987.

[KBCG88]   Kim, W., Ballou, N., Chou, H., Garza, J. F., and Woelk, D., "Integrating an Object-Oriented Programming System with a Database System", *OOPSLA '88 Conference Proceedings*, September 1988.

[KL89]   Kim, W., and Lochovsky, F. H., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, 1989.

[LHR88]   Lieberherr K., Holland, I. and Riel, A., "Object-Oriented Programming: An Objective Sense of Style", *OOPSLA '88 Conference Proceedings*, September 1988.

[PW90]   Pinson, L. J. and Wiener R. S., *Application of Object-Oriented Programming*, Addison-Wesley Publishing, 1990 .

[Rumb87]   Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language", *OOPSLA '87 Conference Proceedings*, October 1987.

[SZ87]   Smith, K. E., and Zdonik, S. B., "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems", *OOPSLA '87 Conference Proceedings*, October 1987.

[Snyd86]   Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", *OOPSLA Conference Proceedings*, October 1986.

[WW89]   Wirfs-Brock, R. and Wilkerson, B., "Object-Oriented Design: A Responsibilty Driven Approach", *OOPSLA '89 Conference Proceedings*, October 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Computer Science Dept. Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
4.	Maj M. L. Nelson, USAF, Code CS/Ne Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
5.	C. Thomas Wu, Code CS/Wq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
6.	Dr. A. Orooji Computer Science Department University of Central Florida Orlando, FL 32816	1
7.	Stephen C. Filippi, LT/USN 101 Bantry Drive Lake Mary, FL 32746	3

GAYLORD S